SPPU New Syllabus

A Book Of

Principles of Programming Languages

For M.Sc. (Computer Science) : Semester - I

[Course Code CS-503-MJ : Credits-02]

CBCS Pattern

CBCS As Per New Syllabus (Under NEP), Effective from 2023-2024

Mrs. Dipali Mali (Mahajan)
MCA (Science), CBSE-UGC NET, Pursuing Ph.D.
Asst. Professor
PDEA's Annasaheb Magar College, Hadapsar
Pune 4110028

Mrs. Monica Deshmane (Dhotre)
MCS, UGC NET, Pursuing Ph.D.
Asst. Professor
Haribhai V. Desai College of Commerce, Arts and Science
Pune 411002

Price ₹ 250.00

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

N1166

PRINCIPLES OF PROGRAMMING LANGUAGES **ISBN 978-81-19116-41-6**

First Edition : September 2023

Published By :
NIRALI PRAKASHAN
Abhyudaya Pragati, 1312, Shivaji Nagar,
Off J.M. Road, Pune – 411005
Tel - (020) 25512336/37/39
Email : niralipune@pragationline.com

Polyplate

Printed By :
YOGIRAJ PRINTERS AND BINDERS
Survey No. 10/1A, Ghule Industrial Estate
Nanded Gaon Road
Nanded, Pune - 411041

DISTRIBUTION CENTRES

PUNE

Nirali Prakashan
(For orders outside Pune)
S. No. 28/27, Dhayari Narhe Road, Near Asian College
Pune 411041, Maharashtra
Tel : (020) 24690204; Mobile : 9657703143
Email : bookorder@pragationline.com

Nirali Prakashan
(For orders within Pune)
119, Budhwar Peth, Jogeshwari Mandir Lane
Pune 411002, Maharashtra
Mobile : 9657703145, 9890997937
Email : niralilocal@pragationline.com

MUMBAI
Nirali Prakashan
Rasdhara Co-op. Hsg. Society Ltd., 'D' Wing Ground Floor, 385 S.V.P. Road
Girgaum, Mumbai 400004, Maharashtra
Mobile : 7045821020, Tel : (022) 2385 6339 / 2386 9976
Email : niralimumbai@pragationline.com

DISTRIBUTION BRANCHES

DELHI
Nirali Prakashan
Room No. 2 Ground Floor
4575/15 Omkar Tower, Agarwal Road
Darya Ganj, New Delhi 110002
Mobile : 9555778814/9818561840
Email : delhi@niralibooks.com

BENGALURU
Nirali Prakashan
Maitri Ground Floor, Jaya Apartments,
No. 99, 6th Cross, 6th Main,
Malleswaram, Bengaluru 560003
Karnataka; Mob : 9686821074
Email : bengaluru@niralibooks.com

NAGPUR
Nirali Prakashan
Above Maratha Mandir, Shop No. 3,
First Floor, Rani Jhanshi Square,
Sitabuldi Nagpur 440012 (MAH)
Tel : (0712) 254 7129
Email : nagpur@niralibooks.com

KOLHAPUR
Nirali Prakashan
New Mahadvar Road, Kedar Plaza,
1st Floor Opp. IDBI Bank
Kolhapur 416 012 Maharashtra
Mob : 9850046155
Email : kolhapur@niralibooks.com

JALGAON
Nirali Prakashan
34, V. V. Golani Market, Navi Peth,
Jalgaon 425001, Maharashtra
Tel : (0257) 222 0395
Mob : 94234 91860
Email : jalgaon@niralibooks.com

SOLAPUR
Nirali Prakashan
R-158/2, Avanti Nagar, Near Golden
Gate, Pune Naka Chowk
Solapur 413001, Maharashtra
Mobile 9890918687
Email : solapur@niralibooks.com

marketing@pragationline.com | www.niralibooks.com

Also find us on www.facebook.com/niralibooks

Preface ...

"Principles of Programming Languages" is a fundamental subject of Computer Science which is designed to become expertise in designing algorithms in most efficient ways by considering its time and space complexity. Due to this student can easily compare code in any programming language. It is our pleasure to present book on this subject for **MSc. (Computer Science): Semester -I as per updated syllabus at 2023 pattern (Under NEP)**

To make subject very interesting, there are lot of comparative language codes are given like Fortran, COBOL, Ada, C, C++, Java, C#, ML and so on. Each chapter contains simple language for explanation and short code or examples are given with output. This book covers various types of scope rules, bindings, control flow statements, data types in different languages, subprograms and nesting, also it covers OOPS concepts like abstraction, encapsulation, inheritance and polymorphism. At the end of each chapter, Multiple Choice Questions, Short answer questions and Long answer questions are provided for practice.

I, Dipali Mali dedicate my work to my parents, all my B.C.S. and M.C.A. teachers, Husband and My daughter Swara.

I, Dipali Mali thankful to my parents Dipti and Deepak Mali, my husband Shailesh Mahajan and my teachers and guide Dr. S.N. Shinde, Dr. Vilas Wani and Dr. Prashant Mulay, for their support and inspiration.

I Monica Dhotre thankful to my mother Mrs. Meera Deshmane, My Husband Sagar Dhotre, my son Smaran, my whole family members, my friends, colleagues who are constant source of inspiration.

I express my sincere thanks to our former principal Dr. Ganesh Raut sir, Dr. Gulab Gugale sir (Chemistry dept H.V.Desai College Pune) for their support in writing book.

Our sincere thanks to Shri. Dineshbhai Furia, Mr. Jignesh Furia for trusting and giving this opportunity to work with them.

We also thank to Mrs. Anita Panajkar, Ms. Madhuri Gabale, Mr. Akbar Shaikh, and Ms. Chaitali Takale for their excellent co-operation.

We also thank to Nirali Prakashan for bringing out the book in market within a short period of time.

Any suggestions and improvements in contents of book will be appreciated and accordingly modifications can be done in later editions.

Asst. Prof. Monica Deshmane (Dhotre)
Haribhai V. Desai College, Pune 2

Asst. Prof. Dipali Deepak Mali (Mahajan)
Annasaheb Magar College, Pune 28

■■■

Syllabus ...

1. Introduction **[2 Hrs.]**

1.1 The Art of Language Design

1.2 The Programming Language Spectrum

1.3 Why Study Programming Languages?

1.4 Compilation and Interpretation

1.5 Programming Environments

2. Names, Scopes, Bindings, Object Orientation Concepts **[6 Hrs.]**

2.1 The Notion of Binding Time

2.2 Object Lifetime and Storage Management

2.3 Static Allocation, Stack-Based Allocation, Heap-Based Allocation, Garbage Collection Scope Rules

2.4 Static Scoping, Nested Subroutines, Declaration Order, Dynamic Scoping The Meaning of Names in a Scope

2.5 Object-Oriented Programming

2.6 Encapsulation and Inheritance, Modules, Classes, Nesting (Inner Classes), Type Extensions, Extending without Inheritance

2.7 Initialization and Finalization, Choosing a Constructor, References and Values, Execution Order, Garbage Collection

2.8 Dynamic Method Binding

2.9 Virtual and Non-Virtual Methods, Abstract Classes, Member Lookup, Polymorphism, Object Closures

2.10 Multiple Inheritance, Shared Inheritance, Mix-In Inheritance

2.11 Semantic Ambiguities, Replicated Inheritance

3. Data Types **[8 Hrs.]**

3.1 Introduction

3.2 Primitive Data Types

3.3 Numeric Types : Integer, Floating point, Complex, Decimal, Boolean Types, Character Types

3.4 Character String Types

3.5 Design Issues, Strings and Their Operations, String Length Operations, Evaluation, Implementation of Character String Types

3.6 User defined Ordinal types Enumeration types, Designs Evaluation Subrange types, Ada's design Evaluation Implementation of user defined ordinal types

3.7 Array types

3.8 Design issues, Arrays and indices, Subscript bindings and array categories, Heterogeneous arrays, Array initialization, Array operations, Rectangular and Jagged arrays, Slices, Evaluation, Implementation of Array Types

3.9 Associative Arrays: Structure and Operations, Implementing Associative Arrays

3.10 Record Types: Definitions of Records, References to Record Fields, Operations on Records, Evaluation, Implementation of Record Types

3.11 Union Types: Design issues, Discriminated versus Free unions, Evaluation, Implementation of Union Types

3.12 Pointer and Reference Types: Design Issues, Pointer Operations, Pointer Problems, Dangling Pointers, Lost Heap Dynamic Variables, Pointers in C and C++, Reference Types, Evaluation

3.13 Implementation of Pointer and Reference Types

3.14 Representation of Pointers and References Solution to Dangling Pointer Problem Heap Management

4. Control Flow **[6 Hrs.]**

4.1 Expression Evaluation, Precedence and Associativity, Assignments, Initialization, Ordering Within Expressions, Short-Circuit Evaluation

4.2 Structured and Unstructured Flow, Structured Alternatives to goto Sequencing

4.3 Selection - Short-Circuited Conditions, Case/Switch Statements Iteration

4.4 Iteration - Enumeration-Controlled Loops, Combination Loops, Iterators, Logically Controlled Loops Recursion

4.5 Recursion - Iteration and Recursion, Applicative- and Normal-Order Evaluation

5. Subprograms and Implementing Subprograms **[5 Hrs.]**

5.1 Introduction

5.2 Fundamentals of Subprograms

5.3 Design Issues for subprograms

5.4 Local Referencing Environments

5.5 Parameter-Passing Methods

5.6 Parameters that are Subprograms

5.7 Overloaded Subprograms

5.8 Generic Subroutines, Generic Functions in C++, Generic Methods in Java

5.9 Design Issues for Functions

5.10 User-Defined Overloaded Operators Coroutines

5.11 Implementing Subprograms

5.12 The General Semantics of Calls and Returns

5.13 Implementing "Simple" Subprograms

5.14 Implementing Subprograms with Stack-Dynamic Local Variables

5.15 Nested Subprograms Blocks

5.16 Implementing Dynamic Scoping

■■■

Contents ...

■■■

CHAPTER 1

Introduction

Contents

Objectives ...

After learning this chapter you will be able:

- To learn about syntax from semantics of Programming Languages.
- To understand comparative study of programming language designs.
- To understand the role of compiler and Interpreter in the programming languages.
- To do study of Programming environment.

1.1 INTRODUCTION

- Programming language is any notification for a description of algorithms and data structures.
- It is a notational system that describes computation in machine-readable and human-readable forms.
- It defines a set of instructions compiled together to perform a specific task by the CPU (Central Processing Unit).

1.1.1 Types of Programming Languages

1. **LLL (Low-level Language):** It is very close to writing actual machine instructions. Languages like assembly languages are designed to more closely resemble the computer's instruction set than anything that is human-readable are low-level languages.
   Example: Assembly Language.
2. **HLL (High-level Language):** HLL is closer to human languages. It is user-oriented, designed to convert an algorithm into program code.
   Example: C++, C#, Python.
3. **MLL (Machine-level Language):** MLL contains a set of binary instructions. Languages direct are readable by machine. It is not portable as computer has its machine instructions, so if a program is written on one computer will no longer be valid on another computer.
   Example: C.

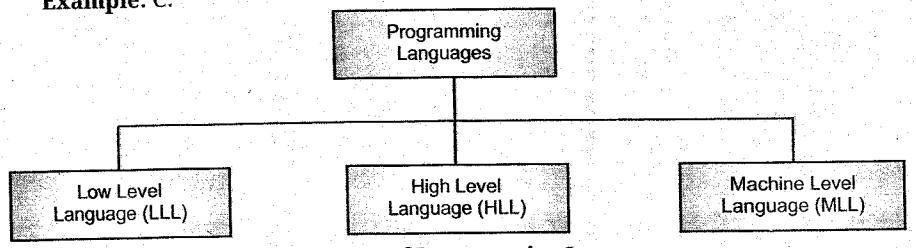


Fig. 1.1: Types of Programming Languages

1.2 THE ART OF LANGUAGE DESIGN

- Programming language design is a complex and multifaceted discipline that requires both creativity and technical knowledge. A well-designed programming language can make coding more efficient, increase productivity, and improve software quality.

1.2.1 Why are many varieties of Programming Languages?

- Following are reasons why *there are thousands of languages*:
  1. **Evolution:** Due to revolutions in structured programming language after 1960s, different languages are invented. Structured programming language changed to object-oriented language.

  2. **Special purposes:** Numbers of languages are designed in a specific domain. For example, language for low-level system programming, PROLOG for logical reasoning, LISP as function-based language.
  3. **Personal preference:** Personal preference is nothing but to design own language according to choice.
- Following are **reasons that makes a Programming language successful**:
  1. **Expressive power:** The purpose of it is to make language easy to understand, write and maintain code.
  2. **Ease of use for beginner:** Language should be easy to learn and grasp. For example, PASCAL.
  3. **Ease of implementation:** This makes it easy to implement language in a compact and portable way. Basically, language is successful when it could be implemented easily on small machines, and with limited resources.
  4. **Excellent Compilers:** Compilers should be good. For example, FORTRAN. Some language like LISP has supporting tools for easy and fast compilation.
  5. **Economics, Support and Inertia:** Although there are other languages available, many are preferred due to the installed software and the knowledge of programmers. For example, COBOL and ADA.

1.2.2 Attributes of Programming Languages

- While designing a language, give attention to the following factors:
  1. **Clarity, Simplicity, Unity:** A Programming language defines how to think about algorithms and how to express those algorithms. It should define a clear, straightforward and consistent set of concepts that are used as primitive in the creation of algorithms.
  2. **Orthogonality:** It refers to attribute of being able to combine various features of languages in all possible combinations with every combination becomes meaningful. Orthogonality is the ability to combine different characteristics of languages in every possible combination with every possible combination becomes orthogonal.
  3. **Naturalness for the application:** A language needs syntax to be used properly which reflects the logical structure of the algorithm.
  4. **Support of abstraction:** There is a huge difference between abstract data structure and operations, which describes the solution to the problem, and primitive data structure and operations built in a language.
  5. **Ease of verification:** There are several techniques that verify a program performs its required function correctly.
  6. **Cost of use:** Cost of use depends on the designing cost, implementation cost, testing and debugging cost and so on.

7. **Program Portability**: A language which is widely available and its definition are independent of the features of machine forms a base for the production of transportable programs.
8. **Programming Environment**: If you have a good programming environment, you may find it easier to work with a language that is technically weak than one that is strong but lacks external support.

1.3 PROGRAMMING PARADIGMS/ THE PROGRAMMING LANGUAGE SPECTRUM

- Paradigms are important because they define a programming language and how it works.

Programming Paradigm:

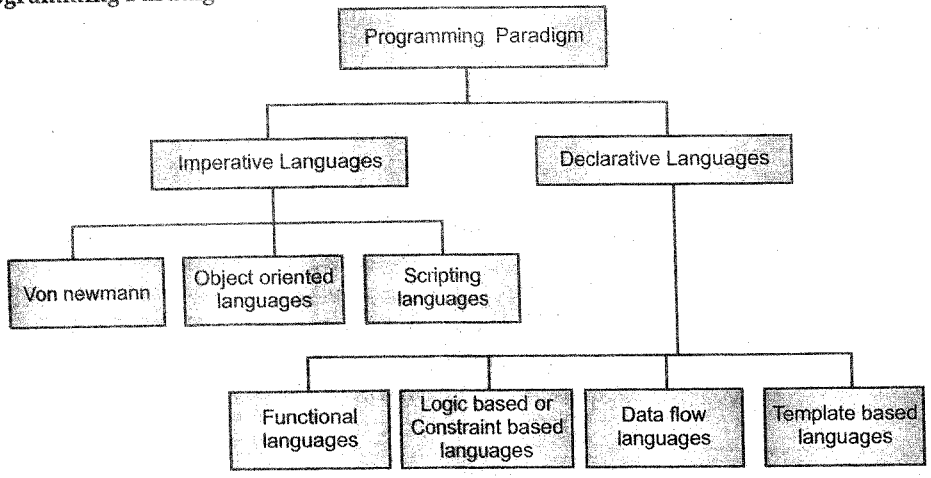


Fig. 1.2: Types of Programming Paradigms

1.3.1 Imperative Languages

- It is designed from implementer's point of view i.e. for performance.
- Focus is on how the computer should do the task?

Types:

1. **Von Newmann Languages**: These are the basic types of languages. Example: C, Pascal, Fortran
2. **Object-oriented Languages**: These languages use object-oriented concepts. It is like Von Newmann but having a more structured and distributed model of memory and computation. Example: C++, Java, C#, Smalltalk.
3. **Scripting Languages**: These languages are used to specify the layout of information in Web documents. Example: PHP, Perl

1.3.2 Declarative Languages

- It has been designed from the programmer's point of view.
- Focus on the performance of the computer.

Types:

1. **Functional languages**: These languages use a computational model based on the recursive definition of functions. These are implemented from the lambda calculus.
   Examples: LISP, ML, Haskell
2. **Logic-based or Constraint-based languages**: PROLOG is known as the best logic based language where the relationship is specified using goal rules.
   Examples: Prolog, Excel, Lotus
3. **Data flow languages**: It is token-based language.
   Examples: Ada, Id, Val, Modula, Fortran
4. **Template-based languages**: The language in which the templates are written is known as a template language.
   Example: XSLT is a template processing model designed by W3C.

1.4 WHY STUDY PROGRAMMING LANGUAGES?

Reasons to study programming languages:

1. **To become the best software engineer**:
   (a) Understand how to use language features.
   (b) Appreciate implementation issues.
2. **Better background for language selection**:
   (a) Familiar with a range of languages.
   (b) Understand issues / advantages / disadvantages.
3. **Good ability to learn languages**:
   (a) You might need to know a lot.

Reasons to study multiple programming languages:

1. Better understanding of implementation issues means Better use of existing programming language.
2. To make it easier to learn and design new languages.
3. To improve the ability to develop effective algorithms.
4. To increase the vocabulary of useful programming constructs.
5. Improved background of programming language selection.
6. To allow better choice amongst alternatives.
7. For understanding obscure features of languages.
8. To make good use of debuggers, linkers and so on.

1.5 COMPILATION AND INTERPRETATION

1.5.1 What is Compilation or Translation?

- **Translation**: Translations perform a 'machine translation' of source code. Translation does not perform a deep analysis on the syntax/ semantics of the code.
- **Compilation**: Compilation does a thorough understanding and translation of the code.
- A compiler/translator changes a program from one language into another.

Examples:

- C compiler converts code from C into Assembly language.
- An assembler then translates it into Machine language.
- Java compiler converts Java code to Java bytecode.
- The Java interpreter then runs the bytecode.

1.5.1.1 Compilation Phases

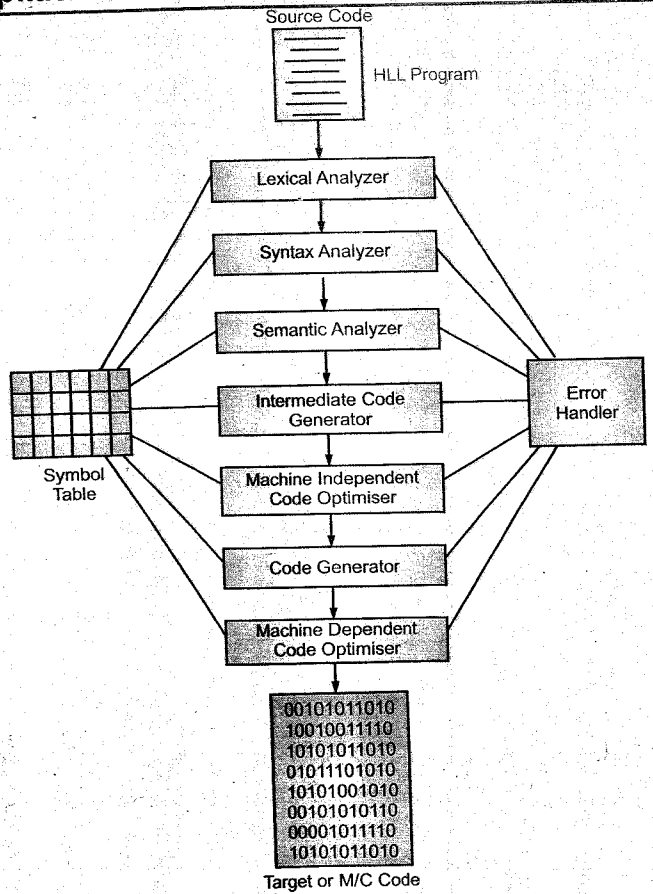


Fig. 1.3: Stages of Compilation

- Let us see details about the Compilation stages.
  - **Lexical Analyzer/Scanner**: This phase scans the source code as a stream of characters called tokens and converts it into meaningful lexemes.
  - **Syntax Analyzer/Parser**: It takes the token given by lexical analysis as input and generates a parse tree (or syntax tree). It checks syntactically correctness for statements.
  - **Semantic Analyzer**: A semantic analyzer checks whether the parse tree is generated is meaningful or not. That means it follows rules of language or not.
  - **Intermediate Code Generator /Variant**: An intermediate code of the source code generated which easily translate into the target machine.
  - **Code Optimizer**: It removes unwanted or dead code lines, and arranges a sequence of statements in correct order to make the program execution fast and without wasting memory.
  - **Code Generator**: The code generator translates the intermediate code into a sequence of (generally) re-locatable (according to the location where it is stored) machine code.
  - **Symbol Table**: All the variable names along with their types are stored here. The symbol table makes it easier for the compiler to fast search the identifier record and retrieve it.
  - **Error handler**: It detects each error, reports it to the user. Then make a recovery plan and implement them to handle the error.

1.5.1.2 Compilation Types

1. **Type 1 - Traditional / Conventional / Pure compiler:**

- No other program is needed to transfer HLL to MLL.

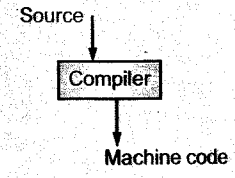


Fig. 1.4: Type 1 - Traditional/Conventional/Pure Compiler

2. **Type 2 - Mixed Compiler:**

- For example, Java uses compiler as well as interpreter. Following figure illustrates that Java compiler with two phases of compilation. In this fig,
  - JVM is Java Virtual Machine.
  - JIT is a Just-In-Time Compiler.

- Byte code is machine independent intermediate output of Java compiler. It can be interpreted as JIT Compiler and converted to machine language.

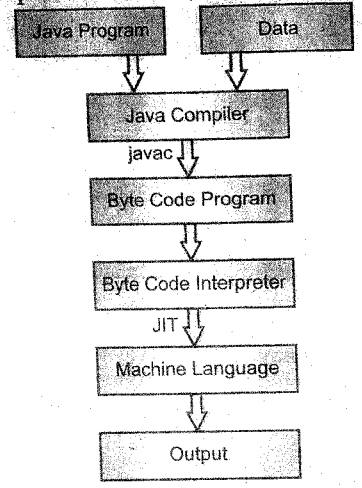


Fig. 1.5: Java Compiler

3. Type 3 - Unconventional / Impure Compiler:

- Compiler generates assembly language instead of machine language code which is also called Post compilation assembly.
- This facilitates debugging as assembly language is easy to learn.

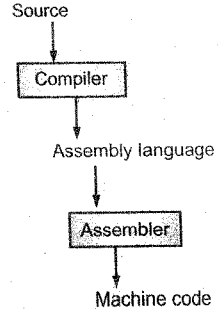


Fig. 1.6: Type 3 - Unconventional / Impure Compiler

1.5.1.3 Examples

1. C Compiler:

- C Compiler compiles source code into machine-language instructions. i.e. Compiler generates assembly language code.

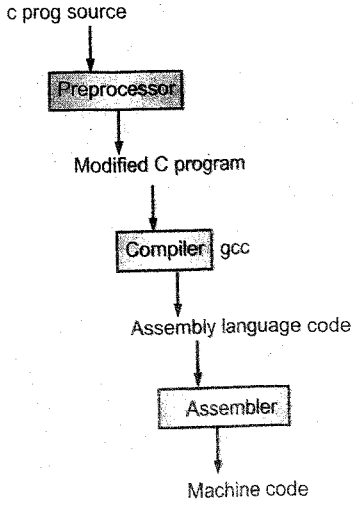


Fig. 1.7: Working of C compiler

2. C++ Compiler:

- C ++ compiler called a true compiler. It performs syntax as well as semantic analysis.
- C++ compiler translates C++ code into equivalent C code, then with the help of C compiler to assembly language code.
- Again that code can be converted to machine language using assembler.

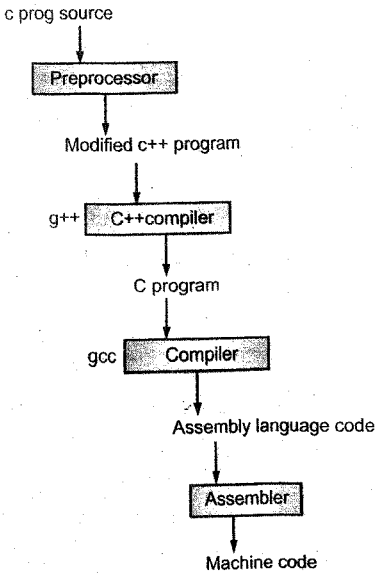


Fig. 1.8: Working of C++ Compiler

1.5.2 Interpretation

- An interpreter is a translator that repeatedly reads one by one instruction and translates them into machine code.

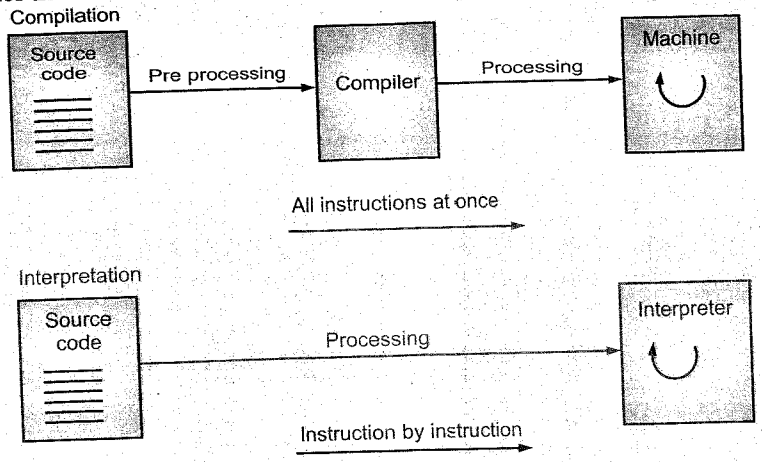


Fig. 1.9: Compiler and Interpreter

1.5.3 Difference between Compiler and Interpreter

Table 1.1: Compiler vs Interpreter

Sr. No.	Compiler	Interpreter
1.	Compiler translates the entire code at once.	Interpreter translates the code line by line.
2.	Users can not see output while translating code.	Users can side by side see output while translating code.
3.	It requires less memory.	It requires more memory compared to the compiler.
4.	It supports static allocation.	It supports static and dynamic allocation.
5.	Though error comes at any line, it proceeds to check other errors.	If an error comes at any line, it halts execution.
6.	Having best performance.	Having somewhat less performance.

1.6 PROGRAMMING ENVIRONMENTS

- In general, programming environment combines software and hardware which allows developer in building applications.
- A programming environment is the collection of tools used in the software development.

- This collection may consist of:
  - A file system.
  - A text editor.
  - A linker.
  - A compiler.
  - Integrated tools.
- These tools may be accessed through an uniform interface (GUI).
- Developers work in integrated development environments (IDEs) mostly. IDEs connect users with all the necessary features required to write and test code correctly. Different IDEs offers different advantages and capabilities.
- IDE provides a convenient workspace for developers when working on a project by packaging all the development tools needed into a single GUI (Graphical User Interface).
- IDEs provide a space for source code editing and a debugging program which locates problems in any written code.
- IDEs offer automation functionality which can handle complaining, packaging, and testing code.
- As various IDE options are available, understanding their general capabilities and attributes is a key in making the right programming environment decision for the project.
- Few **examples of programming environments** are as follows:
  1. **Visual Studio:** Visual Studio is an IDE developed by Microsoft. It is versatile, providing users with a huge library of extensions which allows for more customization than other environments. It is a huge collection of software development tools used through a windows interface. It is used to develop software in languages such as C#, Visual Basic .NET, Jscript (MS JavaScript version), J# (MS Java version), managed C++.
  2. **NetBeans:** NetBeans is an open-source and mainly used for Java projects. It has drag-and-drop user interface as well as number of project templates which make it ideal for all levels of experience. The environment requires extensions be downloaded to be compatible with non-Java projects.
  3. **Eclipse:** Eclipse is an open source IDE which can offer a wide array of functionality to users. It sets itself apart from other environments through the extensive plugin catalog.
  4. **Intellij:** Intellij environment is compatible with many platforms. It offers a strong tool suite to mobile app developers.
  5. **Turbo C and C++:** The Turbo C compiler is a compiler which is used to convert a high level language code into machine code. Turbo C++ is a C++ compiler which is integrated development environment.

6. **Dream weaver:** Dream weaver is a tool which is responsive web design software that is used for site creation and management. Using it we can create complete Websites.
7. **Arduino:** It is an open-source electronic prototyping platform which enables users to create interactive electronic objects. It is able to read inputs using light on a sensor or a finger on a button, or any message and then turn it into an output by displaying activating a motor, LED turn on or publishing something online.

- However, before writing a program, the first thing you need to do is install Environment Setup.
- Environment setup provides a foundation upon which programming can be executed. Therefore, it is necessary to have the software installation on the PC, which will be used for programming, compilation and execution.

Example 1: If you need to access the Internet, then you need to set up the following on your computer:

- A working Internet connection to connect to the Internet.
- A Web browser like Internet Explorer, Mozilla Firefox, Chrome, Safari, etc.

Example 2: To get started with any programming language, you will need to set up the following:

- A text editor to create computer programs.
- A compiler to compile the programs into binary format.
- An interpreter to execute the programs directly.

SUMMARY

- Imperative languages are designed in implementer's point of view i.e. for performance in which focus is on how computer should do tasks.
- Examples of Imperative languages are Von Newmann, Object-oriented languages, Scripting languages.
- Declarative languages designed from programmer's point of view in which Focus is on what computer does.
- Examples of Declarative languages are Functional languages, Logic-based or Constraint-based languages, Data flow languages, Template-based languages.

PRACTICE QUESTIONS

Q.I Multiple Choice Questions.

1. Which is the correct categorization?
   (a) Imperative: Von Newmann, Scripting, Template-based
   (b) Declarative: Logic-based, Dataflow, OOPS
   (c) Declarative: Logic-based, Dataflow, Template-based
   (d) Imperative: Von Newmann, Scripting, Functional

2. Which is the following statement correct?
   (a) Every C code compiled to machine language.
   (b) Every C++ code compiled to assembly language.
   (c) Every C code compiled to assembly language.
   (d) None of these.
3. Machine independent intermediate output of Java compiler is ____.
   (a) Psuedocode (b) Class code
   (c) Byte code (d) Machine code
4. Which code does translation into phases rather than direct source code to machine code?
   (a) Class code (b) Byte code
   (c) P-code (d) Machine code
5. Which data structure in a compiler is used to manage information of variables and their attributes?
   (a) Semantic Stack (b) Abstract Data Type
   (c) Symbol Table (d) Parse Table
6. In which of the following programming language, users write the programs?
   (a) Low-level Language (b) High-Level Language
   (c) Decimal-Format Language (d) Middle-Level Language
7. Which tool is used for grouping of characters in tokens in the compiler?
   (a) Parser (b) Code optimizer
   (c) Code generator (d) Scanner
8. _____ is the collection of tools used in the software development.
   (a) Programming Environment (b) Program tools
   (c) Development Tool-Kit (d) Software Tools

Answers

1. (c)	2. (c)	3. (c)	4. (c)	5. (c)	6. (b)	7. (d)	8. (a)

Q.II Answer the following questions.

1. Which cost is considered while designing language?
2. Write the difference between MLL and LLL.
3. Write the difference between LLL and HLL.
4. Write the difference between HLL and MLL.
5. Write the difference between compiler and interpreter.
6. What is a linker?

7. What is loader?
8. Explain Symbol Table. What is the purpose?
9. What are the three types of compilers?
10. Write examples of Programming Environment.

Q.III Answer the following questions.

1. Explain two unconventional compilers (Hint: C, C++).
2. Write the difference between C and C++ compiler.
3. Explain the phases of compilation.
4. Explain various types of compilers?
5. Explain the Programming language spectrum.
6. What is a programming language? What are types of programming languages?
7. Why are there so many programming languages?
8. What are the attributes of programming languages?
9. Write the difference between Compiler and Interpreter.
10. Describe the types of declarative languages.

CHAPTER 2

Names, Scopes, Bindings, Object Orientation Concepts

Contents

Objectives ...

After learning this chapter you will be able:

- To revise concept of polymorphism, generics , inheritance and overloading.
- To write C++ or any other object oriented language programs very easily.
- To understand difference between scope, lifetime, visibility and binding.
- To learn basic scope rules and differentiate between static and dynamic binding.
- To learn memory allocation and storage.
- To learn how to implement stack.
- To learn basic object oriented concepts in brief.
- To learn designing and implementation of virtual table.
- To study different inheritance types.
- To learn construction and destruction of an object.
- To learn what is dynamic method binding and how to use it.

2.1 INTRODUCTION

- Name is nothing but character string that denotes identifier.
- By reducing the conceptual complexity of the code, gives a better focus on some aspects of a program.
- It provides a level of abstraction in a program, functions for control abstraction, and classes for data abstraction.
- It allows referring entities in a program by a symbol instead of an address.
- There is lot of mechanisms to allocate storage space to this name. According to scoping rules, which we have to learn in brief in this chapter.

2.1.1 The Notion of Binding Time

Binding:

- A binding is an association between two things, such as a name and the thing it represents.
  Example: int x. When this statement is compiled, x is bound to a memory space.

Early / Late Binding:

- Early binding makes the type of the variable. Late binding decided when a value is assigned.
- Function known at compilation time or when the call is being executed to be matched.
- Late binding waits until the value/data assigned to a variable is needed before evaluating or loading it.

Dynamic Binding:

- When the instruction is executed based on context, the exact meaning of each identifier is determined.

Example in Lisp:

- Function X makes reference to a "global" variable p.
- Function Y declares a local variable q and then calls Function X.
- Within that function call, p is the local variable from Function Y.

Binding Time:

- Binding time is the time at which the association between two items is created or the time at which implementation decision is made (compilation, execution, etc.).

Types of Binding Time:

- There are 7 different binding times that can be applied:
  1. Language designing time
  2. Language implementation time
  3. Program writing / editing time
  4. Compiling time
  5. Linking time
  6. Loading time
  7. Run / execution time

- Now we learn each type in brief.

1. **Language design time:**
   - These include control flow structures, primitives, and other semantic statements. Reserved words such as if, for, else, while, etc.
2. **Language implementation time:**
   - These include I/O couplings.
   - It also considers system dependent things such as max heap and stack sizes or arithmetic flow exceptions.
   - Constants: MAX_INT, etc.
   - In Algol, this included the I/O procedure names.
3. **Program writing time:**
   - The programmer's choice of using data structures and algorithms.
4. **Compile time:**
   - The compiler maps high level constructs to machine code.
   - Think of the assembly for a compiled procedure.
   - In this static memory allocation considered.
5. **Link time:**
   - Separate compilation means not every part of a program has to be compiled at the same time.
   - Libraries or header files are linked to your program and bound till that program not completes.
6. **Load time:**
   - Load time is the time required for primitive operating systems to load program variables into memory.
   - Binds physical addresses and in most cases during link time physical addresses maps with virtual addresses.
   - While load time, Relocation of addresses are also considered.
7. **Run time:**
   - It is a very broad term covering the span of whole execution.
   - Values to variable bindings occur at run time.
   - Very important to design and implementation of programming languages.
   - Early bindings considered for greater efficiency.
   - Late bindings considered for greater flexibility.

2.2 OBJECT LIFETIME AND STORAGE MANAGEMENT

2.2.1 Objects LifeTime

- **Object:** It is any entity in the program. Example: a variable, a function.
- **Lifetime:** It is time from creation to deletion of objects.
- **Bindings Lifetime:** The time between creation and destruction of name to obje binding called binding lifetime.

- **Objects Lifetime:** The time between creation and destruction of object called object lifetime.
- Binding lifetime is a subset of object lifetime mostly.
- **Example1:** In pass by reference, the binding dies when you return to function.
- **Example 2:** Dangling reference
  Binding to object that is no longer alive called **dangling reference.**
  Here, Binding is still active, object is gone i.e. A pointer is deleted without making it NULL means contains some value.

```
a = b;
delete a;
b->val;
```

OR

```
delete a;
a->val;
```

- **Example 3:** Memory leak
  Here, binding has gone, but object is still there (memory leak).

```
a=new Node;
a = b;
```

- Example of memory leak is making a pointer NULL without deleting it. Garbage collection gives solution to this problem.

Key events of Objects Lifetime binding:

- Following are several key events occur at name to object binding.
  1. Creation of object
  2. Creation of binding
  3. Reference to variable subroutine time which uses bindings.
  4. Deactivation and reactivation bindings.
  5. Destruction of binding
  6. Destruction of object.

2.2.2 Storage Management

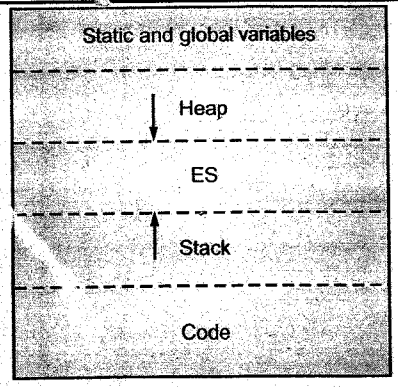


Fig. 2.1: Storage Allocation Mechanism

- To manage the object's space, storage allocation mechanisms are used that shown in the above figure. Following objects are included in the storage allocation Mechanisms.

1. **Static Objects:**
   - Contains absolute storage address that is retained throughout program's execution.
   - Contains Global variables, Program instructions, Static variables, Numeric and String constants. **Example:** "hello".
2. **Stack Objects:**
   - Contains LIFO (Last-in First-out) order for allocation and deallocation, subroutine calls.
   - Contains local variables, arguments, return values, temporaries, saved registers, debugging information.
3. **Heap Objects**
   - Contains allocation/deallocation at arbitrary times means it holds run time memory allocated variables.
   - More expensive.
   - Tables used by run-time support (for debugging, dynamic type checking, exception handing, garbage collection).
   - When these values do not need to be changed at run-time. It often kept in read only memory.

- Now, we will learn each allocation type in detail.

2.2.2.1 Static Allocation

- This region is used to store static, global or own variables.
- The instructions that constitute a program's machine language translation can think o as statically allocated objects.
- The variables that are local to a single subroutine but retain their values from one cal or call to the next their space is statically allocated called static variables.
- Numeric and string-valued constant literals are also statically allocated. For example `printf("hello, world\n")`.
- Statically allocated variables or objects whose value should not change durin program execution (e.g., instructions, constants, and certain run-time tables). Thes are often allocated in protected, read-only memory, so that any attempt to write o update to them will cause an interrupt, which in turn operating system to generate run-time error.
- Local variables are created when function is called, and destroyed when it returns caller function.
- If the subroutine will call repeatedly, each call is said to create and destroy a separat object of each local variable.
- In some languages, named constant needs to be initialized at compile time.

- Along with local variables and elaboration-time constants, the compiler typically stores other information associated with the subroutine, including dynamic link, debugging information, additional saved registers, arguments and return values.
- In FORTRAN, no subroutine recursion and In Basic, no function-level scopes are present.
- Modern compilers keep these temporaries in registers.

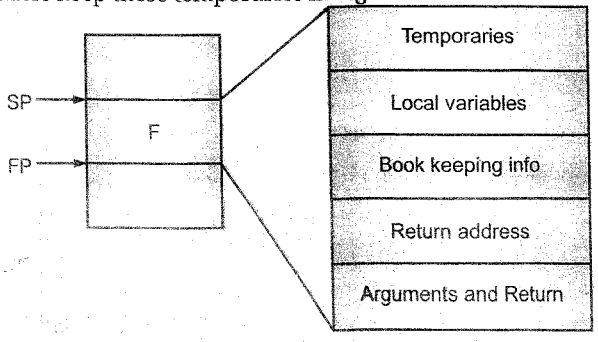


Fig. 2.2: **Static Allocation: Data Memeory Layout**

- **Stack Pointer (SP):** At any given time, the stack pointer register contains the address of last accessed location called top of the stack, or the first location which is unused, depending on convention. Point to location where arguments would be for next call.
- **Frame Pointer(FP):**
  - The frame pointer register contains an address within the frame.
  - It is pointing to start of activation record.
  - Objects in the frame are accessed via frame pointer.
  - If the size of an object is not known at compile time, then the object is placed in a variable-size area at the top of the frame. Its address and dope vector or descriptor stored in the fixed-size part of frame, at offset, which is statically known from the frame pointer.
  - If there are no variable-size or dynamic objects, then every object within the frame has a statically known offset from the stack pointer.
  - If the size of an argument is not known at compile time, then the argument may placed in a variable-size part of the frame after the other arguments, with address and dope vector at known offsets from the frame pointer.
  - Every stack frame contains a reference to the frame of the subroutine that lexically surrounded. This reference is known as the static link.
- At compile time, there are assigned fixed offsets to arguments and local variables from the stack pointer or frame pointer.
- **Dope Vectors (Descriptor):**
  - At compilation, Symbol table maintains dimension and bounds information of each array in the program.

- For every record, it maintaining its offset.
- When the number and bounds of array dimensions are statically known, the compiler can check the symbol table to compute the address of elements of the array. When these values are not known at compile time, the compiler generates code in a dope vector at run time.
- Mostly, a dope vector will contain the lower bound of each dimension and the size of each dimension. If the language implementation semantically checks for out-of-bounds subscripts for array then dope vector may contain upper bounds also.
- See how dope vector looks like :

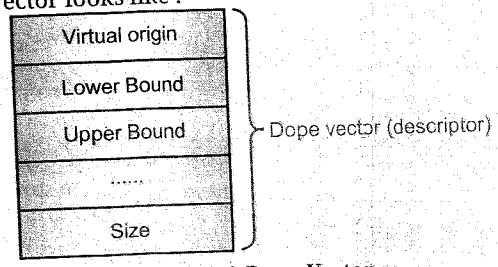


Fig. 2.3: A Dope Vector

- The contents of the dope vector are initialized at run time (elaboration time), or whenever the number or bounds of dimensions change.

2.2.2 Stack-based Allocation (LIFO)

- If a language allows recursion, static allocation of local variables is no longer an option, as number of instances of a variable that may need to exist at the same time is conceptually unbounded.
- So in that case nesting of subroutine calls makes it easy to allocate space for locals on a stack.
- At the run time, each instance of a subroutine has its own separate frame (Also called an **Activation Record** or AR) on the stack, which contains arguments and return values, local variables, temporaries, and book keeping information as shown in the figure.

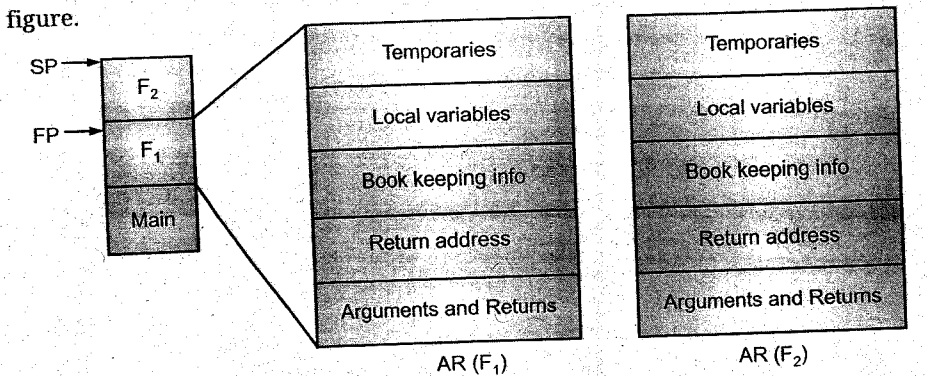


Fig. 2.4: Stack-based Allocation

- An Activation Record stores information about subprogram calls and following data:
  1. **Static Link**: To the static parent (where the subprogram is declared).
  2. **Dynamic Link**: To the caller of this subprogram.
  3. **Temporaries**: Intermediate values generated by calculations.
     Registers are used by compilers to store intermediate values.
     Example:

```
for(int i=0 ;i<n ;i++)
C= i*i*i*i+2;
print C;
```

     Here, intermediate values of C stored to temporaries.
  4. **Local Variables:** These variables declared in scope.

```
F()
{
   int a;
}
```

  5. **Book Keeping Information**: Return address of subroutines, saved registers, debugging information, and the callers stack frame's reference i.e. dynamic link.
  6. **Return Address:**

```
main()
{
   F()
}
F()
{
   Return  ---where to return i.e. to main caller function address
}
```

     Here, function F() will return to caller function address in main ().
  7. **Arguments/Parameters and Returns:**

```
F(int a){} --- argument to function Returns
main()
{
   a=F();  --- return variable
}
F()
{
   return x;
}
```

- In the above example 'a' is argument/parameter of type integer passed to function F(). F() returns x in its definition, whose value is assigned to variable a in main().
- Consider the following code and Trace output:

```
main()
{
   ....
   F();
}
F()
{
   printf("hi");
   F();
}
```

It will print hi till the stack touches to heap or till the stack overflows.

2.2.2.3 Heap Management/ Heap-Based Allocation

- Heap is a storage in which sub blocks can be allocated and de-allocated at arbitrary times.
- Heaps are required for the dynamically allocated data structures, and for objects like fully general character strings, lists, and sets, whose size may change at any point by assignment statement or any other update operation.
- Heap contains free and allocated block list.

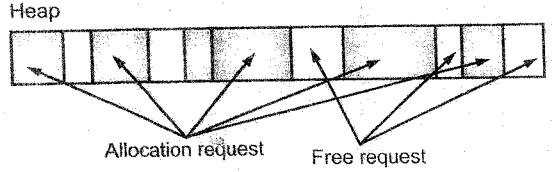


Fig. 2.5: Heap-based Allocation

- List of heap blocks not currently in use called *free blocks*.
- Many storage-management algorithms maintain a single linked list - the free list of heap blocks which are not currently in use.
- If search through list for a block of right size (first fit, best fit) then cost is linear in the number of free blocks.
- Some systems have different lists (pools) for different size blocks.
- For allocating blocks, storage algorithms uses request is rounded up to the standard size. So it causes fragmentation.
- Used for dynamically allocated pieces of data structures. As a result of an assignment statement or update operation objects, Strings, Lists, and Sets size may change.
- One of the risks is external fragmentation. Actually enough space, it is just spread out.

Fragmentation Problem and its solution:

Now we will discuss one by one fragmentation problem and its solution.

1. **Internal Fragmentation:**

- Internal fragmentation happens when the memory is split into mounted sized blocks (Unused space remained in same block). It occurs when a given object allocates a block that is larger than required.

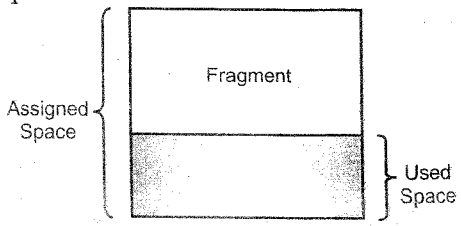


Fig. 2.6: Internal Fragmentation

2. **External Fragmentation:**

- External fragmentation happens when there is sufficient quantity of area within the memory to satisfy the memory request of a method. However the process's memory request cannot be fulfilled because the memory offered is during a non-contiguous manner (unused space remained in different blocks in scattered).

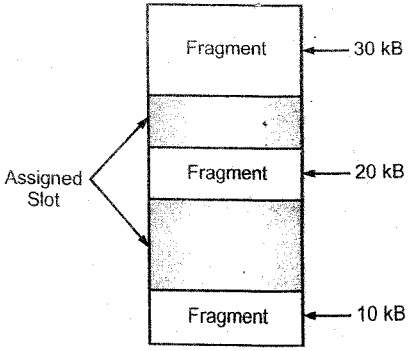


Fig. 2.7: External Fragmentation

Table 2.1: Difference between Internal and External fragmentation

Sr. No.	Internal Fragmentation	External Fragmentation
1.	Fixed-sized memory blocks in the Internal Fragmentation.	Variable-sized memory blocks in the External Fragmentation.
2.	Internal fragmentation happens when the method or process is larger than the required memory.	External fragmentation happens when the method or process is removed.

contd. ...

3.	Internal fragmentation occurs when memory is divided into fixed sized partitions.	External fragmentation occurs when memory is divided into variable size partitions based on the size of processes.
4.	The solution of internal fragmentation is best-fit block.	Solution of external fragmentation is compaction, paging and segmentation.
5.	The difference between memory allocated and required space or memory is called Internal fragmentation.	The unused spaces formed between non-contiguous memory fragments are too small to serve a new process, is called External fragmentation.

3. **Compaction:**

- To eliminate external fragmentation, we must be prepared to compact the heap, by moving already-allocated blocks.

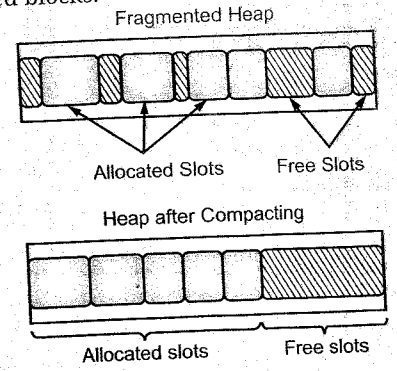


Fig. 2.8: Compaction

Example 1:

- Consider code below, find when binding of x will be there?

```
main()
{
   ….
   F();
}
function F()
{
   int x;
}
```

Output:

There will be binding of x, when x is called.

Example 2:

- Show the memory allocation of code below:

```
int x;
main()
{
   int i;
   int *p=new int;
   F(i);
}
F(int t)
{
   Static int s;
}
```

Memory Allocation for Code is:

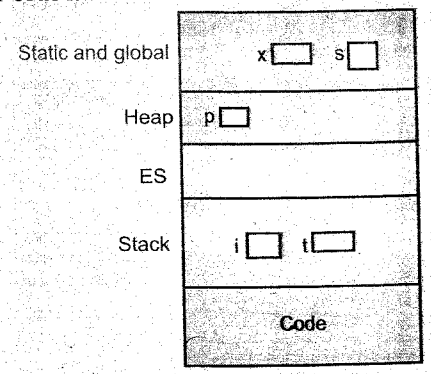


Fig. 2.9: Memory Allocation for Code

2.3 GARBAGE COLLECTION

- Many languages specify that objects are to be deallocated automatically when it is no longer in use. Then the run-time library for such language must provide a garbage collection mechanism to identify and regain memory allocated for these unwanted objects.
- Java uses implicit storage deallocation of objects is called **Garbage collection**.
- If an object is deallocated early, it causes a dangling reference, accessing memory used by another object.
- If an object is not freed at the end of its lifetime, then it causes "memory leak". It also termed as running out of heap space.
- Deallocation errors are extremely difficult to identify and fix.

Lost Heap-Dynamic Variables:

- It is an allocated heap-dynamic variable that is no longer accessible to the program. Such variables are called garbage, as they are not useful and they also cannot be reallocated for some new use.

Example:

- Consider, Pointer p is set to point to a newly created heap-dynamic variable.
- Pointer p is now again set to point to another newly created heap-dynamic variable.
- The first variable p is now inaccessible, or lost which is called memory leakage.

2.3.1 Techniques to collect Garbage Explicitly

- There are various techniques to collect garbage explicitly which are listed below:

1. **Mark and Sweep:**
   - In this method,
     - **Step 1** : Mark all list memory locations or cells as unreferenced or garbage.
     - **Step 2** : Start with known references, (global variables, locals,) mark the cells it references as KEEPERS.
     - **Step 3** : Mark the cells those refer to unreferenced or garbage (stop when you encounter a cell already marked).
     - **Step 4** : Sweep through memory means link all unreferenced cells to the free list.
2. **Reference Counting:**
   - In this method,
     - **Step 1** : Each object has associated a count for the number of references to it.
     - **Step 2** : Garbage is identified when reference count reaches to zero.
     - **Step 3** : An object's reference count is incremented when a reference to it is created and decremented when a reference is destroyed.
     - **Step 4** : When the count reaches zero, the object's memory is reclaimed.

2.3.2 Advantages of Garbage Collection

1. Garbage Collection gives solution to dangling pointer.
2. It avoids problem like memory leakage.
3. Double free bugs occur when the program attempts to free memory that has already been freed and may have been reallocated.

2.3.3 Disadvantages of Garbage Collection

1. Even though the programmer may already know this information, garbage collection still uses a lot of computing resources to figure out which memory to free.
2. The time at which the garbage is actually collected can not be predictable, resulting in scattered stalls throughout a session. Unpredictable stalls can not be acceptable.

2.4 SCOPE RULES

2.4.1 Scope, Visibility, and Lifetime

- The **Lifetime** of a variable is the interval of time in which storage is bound to the variable.

- The action which acquires storage for a variable is known as **Allocation**.

Allocation Types:

1. **Static allocation**: Some languages allocate storage before run-time means at compile time. This is called Static Allocation.
2. **Dynamic Allocation**: When languages allocate storage at run-time either using explicit requests (*malloc, new*) or automatically upon entering a variable's scope means at elaboration time. This is called Dynamic Allocation.

- Languages may use both type of allocation methods.

Scope of variable:

- The scope of variables can be *global* or *local*.
  - A *global variable's* scope includes each and every statement in a program
  - The scope of a *local variable* contains statements only inside the function in which it is declared.
- The similar identifier can be reused inside different function definitions to treat different variables.
- A name is local if it is declared in the current scope, and it is global if declared in an outer scope.

Example:

- Consider the following program with Functions and Parameters.

```
package K
   {
      int h, i;
      void A(int x, int y)
      {
         int i, j;
         B(h);
         ...
      }
      void B(int w)
      {
         int j, k;
         i = 2*w;
            w = w+1;
         ...
      }
      void main()
      {
         int a, b;
         h = 5; a = 3; b = 2;
         A(a, b);
         ...
      }
   }
```

- In the above program,
  - Variables i and j and also parameters x and y are local to function A.
  - Variables declared in function B and main are invisible inside A.
  - The variables h and i are global.
  - Variable h is visible inside function A but i overridden by the local declaration of i.
- The definition of scope differs from language to language.
  - Some languages allow nested type scopes in which every block or statement can contain declarations.
  - Some allows anywhere declarations of variables within a block, while some only previous to any executable statements.
  - Some allow function declarations to be nested (Pascal allows but C does not).
  - Some are more complicated with some special rules for some different types of statements like in Ada.

Examples:

- C/C++ allows declarations to be present in statements;

```
if (a[j] > a[k])
 {
    int t = a[j];
    a[j] = a[k];
    a[k] = t;
 }
 …
 for (int i=0; i<10; i++)
 {
     sum = a[i];
 }
 …
 j = i;
```

- Early languages required that a locally defined identifier be unique in its scope (FORTRAN, Pascal, and C).
- Later languages permitted names to be *overloaded*, allowing multiple definitions of the same identifier within the same scope (Ada, C++).
- This is allowed as long as references can be resolved from the context the identifier is used in.
- Object oriented languages rely deeply on overloading.

2.4.2 Introduction - Scope

- The scope of a variable is region in which the variable is visible.
- A variable is visible in particular statement if it is referenced in that statement. The scope rules of a language determine how a particular occurrence of a name is bound with a variable, or in the case of a functional language, how a name is associated with an expression.

- In simple words, scope rules determine how references to variables are to be declared outside the current executing subprograms or blocks are associated with their declarations and attributes.
- An understanding of rules for a language is very important to write or read programs in that particular language.
- A variable is local in a program unit or block if it is declared there in same scope.
- The non-local variables of a program subroutine or block are those are visible within program subroutine or unit or block but are not declared in that scope means the term non-local is referred for Global variables.

Referencing Environment:

- A Referencing Environment is the complete set of bindings in effect at a given point in a program.
- It means at any given point in a program's execution, the set of active bindings is called the current referencing environment.
- In some cases, referencing environments also depend on binding rules.

Types:

- There are two scope rules:
  1. Static Scope
  2. Dynamic Scope
- Each category having two variations:
  1. Deep binding
  2. Shallow binding
- Also binding can be Ad-hoc binding.

2.5 STATIC SCOPE

- Referencing environment of static scope dependent on lexical nesting of program blocks.
- Static Scope defines the scope of a variable according to lexical structure of a program.
- Using Static Scope each reference to a variable is statically bound to a particular (implicit or explicit) variable declaration.
- Static scope rules are used by most traditional imperative programming languages.
- Here, all scoping can be determined at compile time.
- Most often this is done with a top-to-bottom scan. Algol 60 introduced the method of binding names to non-local variables called static scoping which used in many imperative languages and many non-imperative languages too. Static scoping is so named because the scope of a variable can be statically determined that is, before execution.

Categories of Static-scoped languages:

- There are two categories of Static-scoped languages:
  1. **Nested Subroutines:**
     - In this category, subprograms can be nested which creates nested static scopes. For example, Ada, JavaScript, Common LISP, Scheme, Fortran 2003+, F#, and Python allow nested subprograms.

     (In the next section, we will see this point in more detail.)
  2. **Scope in which subprograms cannot be nested:**
     - Here static scopes are also created by subprograms but nested scopes are created only by nested class definitions and blocks. For example, the C-based languages.

Example:

- Consider the following C code that uses static variables:

```
void label_name (char *s)
{
   static short int n;
   /* static keyword used so static local variables are initialized
   to zero */
   sprintf (s, "L%d\0", ++n);
}
```

2.5.1 Nested Subroutines

- Nested subroutine is the ability to nest subroutines inside each other. It introduced in Algol 60, including Pascal, Ada, ML, Python, Scheme, Common Lisp, and to a limited extent like in Fortran 90. Other languages, including C and its descendants, allow classes or other scopes to nest.
- Just as the local variables of a FORTRAN subroutine are not visible to other side subroutines, any constants, types, subroutines or variables, declared within a block are not visible outside that block in Algol-family languages.

Example:

- The following code shows Nesting subroutine:

```
Procedure A
{
   Procedure B
   {
      Procedure C
      { }
      Procedure D
      { }
   }
   Procedure E
   { }
}
```

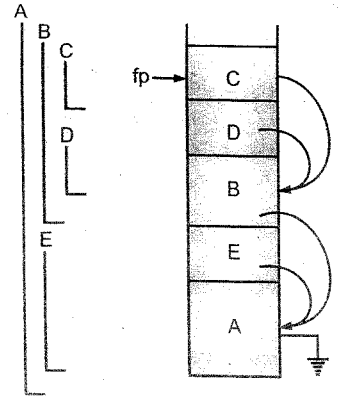



Fig. 2.10: Nested Subroutines and Static Chains

- In the Fig. 2.10, the right side shows Static chains and Subroutines A, B, C, D, and E are nested as shown on the left side.
  - If the sequence of nested calls at run time is A, E, B, D, and C, then the static links or static chain in the stack will look as shown on the right.
  - The code for subroutine C may find local objects at offsets which are known from the frame pointer.
  - It may also find local objects in surrounding scope, B, by dereferencing static chain and then applying its offset.
  - It can find local objects in B's surrounding scope, A, by dereferencing its static chain twice and then applying an offset.

2.5.2 Related Concepts to Static Scope

1. **Problems with the Static Chain:**
   - A non-local reference is slow if the number of scopes between reference and the referenced variable declaration is large. That means if parent chain increases.
   - More access time requires, because the costs of non-local references are not equal, and can change with code upgrades.
2. **Display Registers:**
   - Disadvantage of static chain is that access to an object in a scope k levels from parent to grandparent and so on. It requires that the static chain be dereferenced k times.
   - So one of the alternatives for static chain is to use a display, which is array to store the static links.
   - It uses a pointer array to store the activation records along the static chain. But static chain is better, as display can be kept in registers.

Example:

- Consider nested subroutines above with same calling sequence, (A E B D E), its display registers can be shown with static links as,

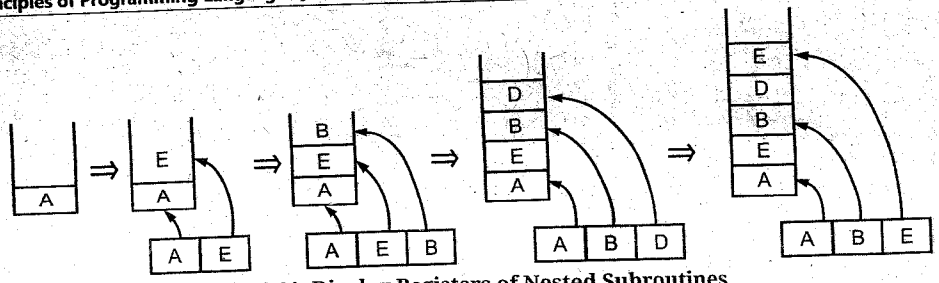


Fig. 2.11: Display Registers of Nested Subroutines

3. **Blocks:**

- Many languages allow new static scopes to be defined within executable code itself.
- This powerful concept initially used in Algol 60 that allows a part of code to have its own local variables whose scope shortened.
- Such variables are typically stack-dynamic, so their storage is allocated when the section is entered and deallocated or freed when the section is exited. Such section of code is nothing but a block.

Example:

- Consider following C code:

```
int main (...)
{ /* start of Block1 */
   float A; int B, C[10];
   .. A;..B;.. C;
   { /* start of Block2 */
      char P; int Q, R[11];
      .. A;..B;..C;..P;..Q;.. R;
   } /* end of Block2 */
      .. A;..B;.. C;
}/* end of Block1 */
```

4. **Module and its Types:**

- Modularization used for information hiding, which makes objects and algorithms invisible, whenever possible, to portion of the system that does not need them.
- The module implementation is compiled separately and implementation details are hidden from the user of the module. Properly modularized code minimizes "cognitive load" on the programmer by reducing the amount of required information to understand.
- There is no language supporting modules in Pascal and C, Modula-2 *modules*, in Ada *packages*, in C++ *namespaces*. In Java class, source files and libraries can be treated as modules.

- A module allows a collection of objects (subroutines, variables, types, and so on) that encapsulated in such a way that:
  1. Objects inside are visible to each other inside.
  2. Objects on the inside are not visible on outer side till not explicitly exported.
  3. In some languages, objects outside are invisible from inside till not explicitly imported.
- Most module-based languages allow the programmer to specify that certain exported names are usable only in restricted ways.
- Variables may be exported as read-only, for example, or types may be exported opaquely.
- A module interface shows exported variables, data types, and also subroutines.

Module Types and Classes:

- An alternative solution to the multiple instance problems can be found in languages such as Simula, Euclid. These languages treat modules as types rather than simple encapsulation constructs.
- Given a module type, the programmer can declare an arbitrary number of similar module objects.
- Euclid also allows the programmer to specify finalization code that will be executed at the end of a module's lifetime.

5. **Opaque Export:**

- Types may be opaquely exported, meaning that variables of that type declared, passed as arguments to the module's subroutines, and compared or assigned to one another, but not manipulated in any other way.

6. **Nested Blocks:**

- In many languages, like Algol60, C89 and Ada, local variables are declared at the beginning of any subroutine, and also at the top of any start i.e. begin... end or brackets ({...}) called block.
- Other languages, including Algol 68, C99, and all of C's descendants are more flexible whereas we can declare wherever a statement may appear.
- In nesting, inner declaration hides outer declaration with the same name.

7. **Closest Nested Scope Rule:**

- A variable declared has its scope in the current subroutine, and in each internally nested subroutine, unless it is hidden by another declaration of the same name.

2.5.3 Declaration Order

- In C89, and in some other languages, all data declarations in a function except having nested blocks must appear at the beginning of the function whereas in some languages such as C99, C++, Java, JavaScript, and C# allow variable declarations to appear anywhere.

- Declarations may create scopes which are not associated with compound statements. For example, in C++, and Java, the scope of local variables is from their declarations to the ends of the block.
- In C#, the scope of any variable declared in a block is the whole block, without considering declaration in the block. In JavaScript, local variables declared anywhere in a function, but the scope of such a variable is always that entire function.
- Though, a variable still must be declared before it can be used.
- In latest versions of C++, the scope of variable is from its definition to the end of the smallest enclosing block.
- Consider the following skeletal method:

```
void myFun()
{...
   for (int count = 0; count < 10; count++)
      {... }
 ... }
```

- In later versions of C++, as well as in Java and C#, the scope of count is within *for* loop only.

2.6 DYNAMIC SCOPING

2.6.1 Dynamic Scope

- Dynamic Scope defines the scope of a variable in terms of program execution. The reference environment depends on the sequence of declarations.
- Each variable declaration affects all subsequent statement execution, until a new declaration for the identifier is encountered.
- Dynamic scope rules are easier to implement but have some limitations too.
- Dynamic scope is most often used by interpreted languages. APL, SNOBOL, early dialects of LISP, and Perl has dynamic scoping.
- Generally following are less complicated to implement.
  - The "current" binding for a name is the one most recently encountered during execution, and not yet destroyed by returning from its scope.
  - Overloading and optional parameters:
    - Consider, `print_integer` takes an *optional* parameter (one with a default value) that specifies the base.
    - `print_integer` is *overloaded*. One form prints in decimal; the other form takes an additional parameter that specifies the base.
- While using scope rules, following questions raised:
  - Which dynamic scope rules are to be applied?
  - When the function is called to use Shallow (late) binding? (In which the reference environment of a function/procedure is determined when the function is called.)
  - When the reference is created use Deep (early) binding? (In which the reference environment of a function/procedure is determined when the function is created.)

Deep binding with Dynamic Scoping:

- Deep Binding is usually used by default in lexically scoped languages.
- Non-local references are found by searching activation records on dynamic chain.
- Length of chain is not fixed as each activation record instance must contain variable names.
- Here, reference environment of multiple processes created early.

Shallow binding with Dynamic Scoping:

- Dynamically scoped languages tend to use shallow binding.
- It is not difficult to understand why dynamic scoping is not as widely used as static scoping.
- Programs in static-scoped languages are easier to read, are more reliable, and execute faster than same programs in dynamic-scoped languages.
- Names and values are stored in a global table, and space is allocated for every variable name that is in the program.
- When function is called it saves the current value of the variable and substitutes it with the value in its current scope. When the function exits, the value of the variable is stored again.
- Referencing environment of multiple procedures created late means one after other.
- Subroutine Closures is combination of representation of reference environment and reference to subroutine.

2.6.2 Advantages and Disadvantages of Dynamic Scoping

Advantages of Dynamic scoping:

1. More flexibility is achieved using dynamic scoping.
2. Things make very easy using dynamic scoping. For example, in parameter passing, no need to pass parameters if they are present in an outer scope.
3. Often parameters passed from one subprogram to another are just local variables to the caller.

Disadvantages of Dynamic scoping:

1. Less readability or less understandable than static scoping.
2. Poor reliability of execution.
3. Difficult to debug as data is passed at run time.
4. No locality of accessing data.
5. No protection to local variables.
6. We can get unexpected results while using Dynamic scoping.

2.6.3 Symbol Table

- In a language with dynamic scoping, an interpreter (or the output of a compiler) must perform operations to symbol table insert and lookup at runtime.
- Symbol table in a compiler could be used to track name-to-object bindings in an interpreter, and vice versa.

- During compilation, symbol table keeps track of all the identifiers.
- Basic functions used are `lookup()` and `insert()`.
- Must keep track of scope visibility by using `enter_scope()` and `leave_scope()` functions.
- Symbol table keeps information about dimension and bounds for each array in the program. For each record, it keeps information about the offset of each field.
- Symbol table contains fields like, index, Symbol name, its type and scope.
- To translate code to assembly or machine language, the code generator in phases of compilation, traverses the symbol table for assigning locations to the variables, and then the intermediate code generated.

2.6.4 A - List and Central Referencing Table - Figure

- In practice, implementations of dynamic scoping tend to adopt one of two ways: An association list or a central reference table.

1. **An association list(or in short called A-list):**
   - It is simply a list of name/value pairs.
   - When used to implement dynamic scoping the functions pushed on stack.
   - New declarations are pushed which creates activation records and popped at the end of the scope.
   - Bindings are found by searching down the list from the top.
2. **Central Referencing Table (CRT):**
   - Access to non-local variables in a dynamic-scoped language can be implemented by using dynamic chain or through some central variable table method.
   - Dynamic chains provide slow accessing but fast calling and returns whereas the central table methods provide fast accessing but slow calling and returns.
   - A central reference table avoids the need for linear-time search by maintaining an explicit mapping from names to their current meanings. Lookup is faster than symbol table, but scope entry and exit are increasing more complexity, and it becomes very difficult to save a referencing environment for future use.

Example:

- Create A-list and central referencing environment for the following code:

```
I, J: Integer
proc P( I: Integer)
…
proc Q
   J: Integer
   …
   P( J )

main
   …
   Q();
```

- Here, P and Q are procedures; I and J are two global variables. Procedure P contains as parameter and Procedure Q contains J as local variable.

- A-list for above code can be prepared as,

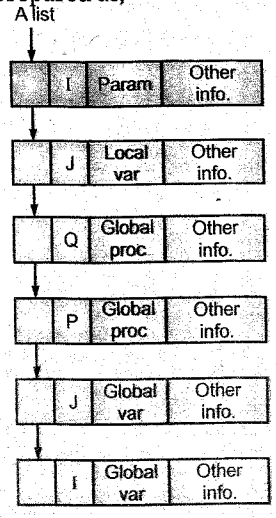


Fig. 2.12 A-List

- Central Referencing Table (CRT) can be shown as,

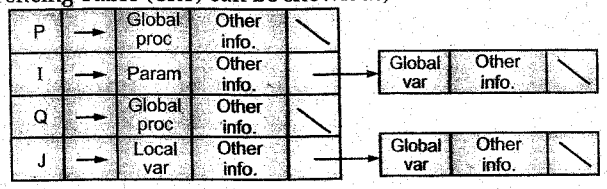


Fig. 2.13 CRT

2.6.5 Related Concepts to Dynamic Scope

1. **Calling Sequence:**

- Consider nesting below.

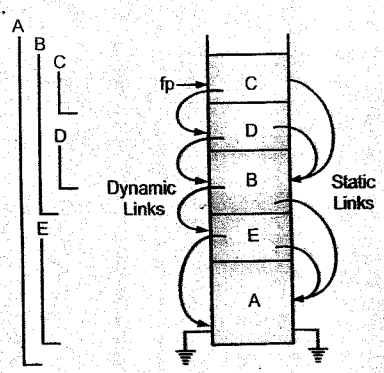


Fig. 2.14: Nesting and Calling Sequence

- Consider calling sequence in which one function calls other is AEBDC.
- In this sequence, A calls E, E calls B and so on. Dynamic links are given as per calling sequence.
  - **The Prologue Actions:**
    (i) Push current BP in the stack as the dynamic link and create the new value.
    (ii) Replace FP /BP with SP (stack pointer).
    (iii) Allocation of local variables.
  - **The Epilogue Actions:** These actions are reverse of prologue actions.
    (i) If pass-by-value-result or out-mode parameters method used, the current values of those parameters moved to corresponding actual parameters.
    (ii) If it is a function, the functional values are moved to that place which is accessible to the caller and replace SP with FP.
    (iii) Pop BP from stack and Restore SP by value of the current EP minus one and set the EP to the old dynamic link.
    (iv) Restore the status of execution for caller.
    (v) Transfer control again to the caller.

2. **Elaboration:**

- The process by which declaration becomes active.
- Elaboration tasks are: Creation of bindings, Allocation of stack, Fill possible arguments with initial values.

3. **Difference between static and dynamic scope or Difference between early and late binding:**

Table 2.2: Difference between Static and Dynamic Binding

Sr. No.	Static Binding	Dynamic Binding
1.	Any binding occurring prior to run-time i.e. compiler or linking time is static binding.	Any binding occurring at run-time i.e. loading or run time is dynamic binding.
2.	Actual object is not used.	Actual object is used.
3.	Binding happens during compilation so called early binding.	Binding happens at execution so called late binding.
4.	Static binding is shown by private, static and final methods as can not be overridden.	Excluding private, static and final methods, other methods which can be overridden shows dynamic binding.
5.	Decisions are associated with static typing (faster code) in earlier binding.	Decisions are associated with dynamic typing, pointers point to different objects (more flexible code) in later binding.

contd. ..

6.	Compile Time Polymorphism also called Early Binding or Overloading or Static binding.	Run Time Polymorphism also called Late Binding or Overriding or Dynamic binding.
7.	It is having greater efficiency.	It is less efficient.
8.	This type of binding is less feasible.	This type of binding is having greater feasibility.
9.	**Example:** Method Overloading	**Example:** Method Overriding.

2.6.6 Implementing Binding

- Consider code below. In this code, we try to solve binding with using both Static and Dynamic scoping rules.

```
x:integer
procedure set_x(n:int)
   x=n
procedure print_x
   write_int(x)
procedure Foo(S, p:function, n:int)
   x:int = 5
   if n in {1,3}
      set_x(n)
   else
      S(n)
   if n in {1,2}
      print_x
   else
   p
set_x(0); Foo(set_x, print_x,1); print_x;
set_x(0); Foo(set_x, print_x,2); print_x;
set_x(0); Foo(set_x, print_x,3); print_x;
set_x(0); Foo(set_x, print_x,4); print_x;
```

Tracing the output:

1. **Static Scoping Rule:**
   - For 1st call, `set_x(0); Foo(set_x, print_x,1); print_x;`

```
set_x(0)
set x's n=0
main's X=0
```

```
Foo(set_x,print_x,1)

main's X=0
Local-
Foo's n=1
Foo's X=5

set_x(1)
main's X=0=1
set x's n=0=1

Foo's n=1
Foo's X=5

Print_x
main's X=1
```

```
Print_x
main's X=1
```

- So, it prints 11 in 1st line of call. Similarly, in second call it prints 22, in third call 33 and in fourth call 44.
- So, output using static scoping rule is:
  11 22 33 44

2. **Dynamic Scoping Rule:**
   - For 1st call, `set_x(0); Foo(set_x, print_x,1); print_x;`

```
set_x(0)
set x's n=0
main's X=0
```

```
Foo(set_x,
print_x,1)

main's X=0
Local-
Foo's n=1
Foo's X=5

set_x(1)
Foo's n=1
Foo's X=5=1
set x's n=1

Print_x
foo's X=1
```

```
Print_x
main's X=0
```

- So, it prints 10 in 1st line of call. Similarly, in second call it prints 20, third 30 and forth 40.
- Therefore, output using dynamic scoping rule is,
  10 20 30 40

2.7 THE MEANING OF NAMES IN A SCOPE

2.7.1 Aliases

- Alias is more than one name refers to same object. Here for each variable value is passed by references.
- Compiler uses Symbol table to differentiate scopes of aliases.

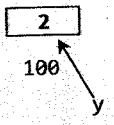
Example 1:

```
double sum, sum_of_squares;
...
void sqrsum (double & x) //x passed by reference
{
    sum =sum + x;
    sum_of_squares += x * x;
}
...
sqrsum (sum);
```

Here, sum and x both are referring same location as aliases.

Example 2:

```
int x=2;
int y=&x
y=3  //x will be 3
x=5  //y will be 5
```



Here, x and y both are referring same location as aliases.

2.7.2 Overloading

- In some programming languages, we can have declaration of same function in many different ways.
- For example, (+) the plus sign is used to define different functions, including signed and unsigned integer types and floating-point in addition. In Java, + used for concatenation and so on.

Types:

- Overloading can have 2 categories:

1. **Function Overloading:**
   Consider the example of C++ code:

```
struct complex
{
    double real, imaginary;
};
```

```
enum base {dec, bin, oct, hex};
    int i;
    complex x;
    void print_num (int n) {...}
    void print_num (int n, base b) {...}
    void print_num (complex c) {...}
    print_num (i);       // uses the first function above
    print_num (i, hex); // uses the second function above
    print_num (x);       // uses the third function above
```

- Here, all print_num functions are overloaded as name of function is same but either number of parameters or data type of parameters are different.

2. **Operator Overloading:**

```
class complex
{
    double real, imaginary;
    ...
    public: complex operator + (complex other)
    {
        return complex(real + other.real, imaginary + other.imaginary);
    }
...
};
...
complex A, B, C;
...
C = A + B;
```

- Here, user-defined operator + is used for overloading.

2.7.3 Polymorphism and Related Concepts

- The term "polymorphic" comes from the Greek, and means "having multiple forms."
- It is applied to code in which both data structures and subroutines that can work with values of multiple types.

Types of Polymorphism:

1. **Parametric Polymorphism:**
   - This polymorphism takes generic parameters which used in type expressions that describe the types of the parameters of subprogram.
   - Types of parameter polymorphism are explicit parametric polymorphism and implicit parametric polymorphism.

   - Examples of Implicit Parametric Polymorphism: Lisp, ML and in the various scripting languages.
   - Example of Explicit Polymorphism: Generics.
2. **Subtype Polymorphism:**
   - Here, the code work with values of some specific type T, but the programmer can define additional types to be extensions of T, and the polymorphic code will work with these subtypes as well.
   - Example, Subtypes (classes) are inherited from the methods of their parent types in the Object-oriented Language.
3. **Ad-hoc Polymorphism:**
   - The Ad-hoc Polymorphism is a technique used to define the same method with different implementations and different arguments.
   - Overloaded subprograms provide a particular kind of polymorphism is called Ad-hoc polymorphism.
   - In a C++, Java programming language, Ad-hoc polymorphism achieved with a function overloading.
   - In this method, binding happens at the time of compilation.
   - Ad-hoc polymorphism is also sometimes referred as compile-time polymorphism.
   - Every function call binds with the respective overloaded method based on the passed arguments.

Coercion:

- Coercion is the process by which a compiler automatically converts a value of one data type into another type when that second type is required for result. For example,
  1. Pascal will coerce integers to floating point in expressions and assignments.
  2. FORTRAN will also coerce floating-point values to integers in assignments, at a potential loss of precision.
  3. C will perform these same coercions on arguments to functions.
  4. Most scripting languages provide built-in coercions.
  5. C++ allows the programmer to have extensions with its built-in set and user-defined coercions.

Generosity/Generics:

- Explicit parametric polymorphism is also known as **generosity**. Generic facilities appear in Ada, C++, C, Java, and C# and so on. In C++, they are known as templates.
- Generics (explicit parametric polymorphism) are usually, Inheritance (subtype polymorphism) is almost implemented by creating a single copy of the code, and by including in the representation of objects," metadata" (data about the data).
- Implicit parametric polymorphism can be implemented either compile or run time.
- Most Lisp implementations checks at run time.
- ML and its next languages perform all compile time type checking.

- Consider **Ada language code for finding maximum** below:

```
type T is private;
with function ">"(a, b : T) return Boolean;
function max(a, b : T) return T;
function max(a, b : T) return T is
begin
   if a > b then return b;
   else return a;
end if;
end max;
function string_max is new max(string, ">");
function date_max is new max(date, date_precedes)
```

- Here, T is template called generic type which varies according to passed parameters.

2.8 OBJECT-ORIENTED PROGRAMMING

- The concept Object-Oriented Programming introduced in SIMULA 67.
- Smalltalk is Object-Oriented Programming which is pure.
- For Object-Oriented Programming there are main three features:
  (a) Abstract Data Types
  (b) Dynamic Binding
  (c) Inheritance

Benefits of Module Abstraction:

1. The conceptual load is reduced by minimizing the amount of details as that the programmer thinks.
2. By preventing the programmer from using program components in inappropriate way, by restricting the part of a program's text in which might use a given component, provides fault containment. When searching for bug's cause, limits the portion to be considered.
3. Provides degree of independence among program components, to make easy to assign construction to separate individuals, to modify internal implementation without change in used external code, or to install in a library where used by other programs.

Methods: Members of subroutine in object-oriented languages.

Fields: Data members.

- In C++, *this* keyword refers to object of currently executing subroutine.

- Consider the following C++ code,

```
class node
{
   node *prev;
   node *next;
   node *head;
public:
   int a;
   node *proc1(){}
   node *proc2(){}
   void insert(node *p);
};
```

- In the above code, node class contains data members `prev`, `next` and `head` as private members and 'a' is public data member.
- Member functions `proc1()`, `proc2()` and `insert()` as public.
- Now consider function `insert()` defined outside class then we have to use scope resolution (::) operator.
- Example, `void node::insert(node *mynode){ }`.

2.9 ENCAPSULATION AND INHERITANCE

2.9.1 Encapsulation

- Encapsulation allows the programmer to **group** data and the subroutines operating together in one place on them and **hides unnecessary details** from an abstraction's user.
- In C++,
  - Data members and Data functions have two variations: class members and instance members.
  - In instance members, if it is class instance it may be allocated as static (accessed by value), stack dynamic, or heap dynamic (accessed by reference and created using new and delete keywords).

Example:

```
class A
{
   public:
   int a, b;
   void display()
   {}
}
```

- In Ada, encapsulation is also known as Package. It contains two parts as follows:
  1. The **Package specification**: Providing interface of encapsulation.
  2. **Body package**: Contains implementation code for package specification. The reserved word **body** used to specify this part.
- The same name shared by both parts. Both parts can be separately compiled. Mostly first the package specification is compiled, then the body package get compiled.

2.9.2 Modules

- In Clu and Euclid, module's declaration and definition (header and body) appears together always.
- Header states which module's names are exported.
- In Euclid, if a module M exports any type T, the remained part of program will be idle with object of T other than passing T to subroutines which are exported from module M.
- Here, T is called an opaque type.

```
var Mymodule : module
exports(t with (:=, name))
 …
type t = record
    var name : array of char
      …
end t
  …
```

- The code outside `Mymodule` module will assign **tuple** variables to one other, accessing their name fields, but equality can not be checked.
- In Euclid, modules may have any number of instances.
- To avoid complications, may hide this parameter to store address of different instances.
- Clu introduced this type of module-based scoping used in 1970's in Euclid, and Modula
- Module has 2 parts header and body, if both parts are in different files; updation to body never requires recompilation of module. But updation to private members of header requires recompilation of module.
- In Java, modules can be expressed as a package which contains different files. Each file created for class in which variables and functions are declared.
- C++ contains namespace for operating as modules.

2.9.3 Classes, Nesting (Inner Classes)

Classes:

- Modules inheritance concept is achieved using base class and derived class.
- **In C++**, visibility can be achieved using public, private and protected keywords.
- Public members can be accessed by any other class instances.

- In C++, to hide base class's details can inherit base class privately.
  - For example, consider the code below,

```
class A: private B
{ };
```

  - Protected class access members within class and its derived classes only.
- **In Java**, C# derived class cannot make private base class.
- **In Smalltalk and Objective-C**, visibility problems may not occur as at run time code may call any method with any object.
- If method with correct parameters exist call processed otherwise run time error triggered.
- **Eiffel** uses export status for visibility.
- Member is private if status is NONE and member is public if status is ANY.

Nesting (Inner Classes):

- To declare a Class inside another class called Nested class.
- Inside class called as Inner class, outside class is Outer class.

Example 1:

- Consider Java's nesting below,

```
class A
{  //outer class
   void displayA()
   {
      B b=new B();
      b.displayB();
   }
   class B      //inner class
   {
      void displayB()
      {}
   }
}
class MyClass
{
   public static void main()
   {
      A a=new A();
      a.displayA();
   }
}
```

- In the above code, class B is nested inside class A. From main method outer class object is created and from outer class inner class object created to access members.

Example 2:

- Consider the following code in Java,

```
class O
{
    into_variable;
    class I
    {
        public void assign()
        {
            o_variable = 1;
        }
    }
    O() // constructor
    {
        I i_variable = new I();
    }
    public void display()
    {
        O_variable = 0;
        System.out.println(o_variable);      // prints 0
        i_variable.assign();
        System.out.println(o_variable);      // prints 1
    }
}
```

- In the above code, object of inner class created in outer class constructor to acces members.
- In C++ and C#, only static members of outer class accessed within inner class as only single instance is created by them which leads to data hiding.

2.9.4 Type Extensions

- Smalltalk, Eiffel, C++, Java, Objective-C and C# all are object-oriented languages thoug having weak encapsulation mechanism. These languages describe module a abstraction for encapsulation and inheritance.
- Some of languages, like Modula-3, Ada 95, FORTRAN 2003, represented as Object oriented languages with by default encapsulation.
- Type Extensions can be done using keyword *extends*. It makes reusable code calle **Inheritance**.

Example:

```
class B extends A
{ ..... }
```


2.9.5 Extending without Inheritance

- For making extensions inheritance is essential.
- We may restrict inheritance. For example, in Java by using *final* keyword and in C#, *sealed* keyword is used.
- In C#, we can use extensions with static class.

Example:

```
static class A{
public static int toInt(this String str)
{
    return Integer.paseInt(str);
}
int n = A.toInt("10");
```

- These extensions not able to access private members which they extend and also dynamic binding is not supported.

2.10 INITIALIZATION AND FINALIZATION

- Many object-oriented languages provide mechanism to initialize an object automatically at the lifetime's beginning.
- This mechanism is known as a constructor when written in a subroutine form.
- A constructor does not allocate space.
- Initialize space which has already been allocated.
- Some languages provide destructor mechanism to finalize an object automatically at the lifetime's end.

2.10.1 Initialization

- Following code shows use of initialization and assignment.

```
class A
{
    Public int a,b;
    void A()
    {
        this.a=10; this.b=20;
    }
    void A(int x, int y)
    {
        this.a=x; this.b=y;
    }
}
```

```
public static void main()
{
   A obj=new A();
   A obj=new A(1,2);
}
```

Difference between Initialization & Assignment:

- Initialization provides initial value to variable which uses assignment or side effect.

```
{
   this.a=10;
   this.b=20;
}
//assignment
void A(int x, int y)
//initialization x=n, y=m
{
   this.a=x;
   this.b=y;
}
public static void main()
{
   int n=1,m=2;     //assignment
   A obj=new A();
   A obj=new A(n,m);
}
```

2.10.2 Four Important Issues

2.10.2.1 Choosing a Constructor

- Object-Oriented Languages allows classes which may contains zero, one or more different constructors.
- Different constructor may have different names or can be distinguished by number and types of arguments.
- There are two ways to distinguish between constructors as follows:
  - Using Different Name
  - Using Different Number and Types of Arguments
- In C++, C#, Eiffel, Java and Smalltalk, for a given class the programmer can specify many constructors.
- In **C++, C# and Java**, the constructor acts like overloaded subroutines distinguished by number and types of arguments.

- In **Smalltalk and Eiffel**, different constructors having different names; code creating an object must name a constructor explicitly.
- For example, consider the following code:

```
Class mydata
{
   public:
   mydata(String data);
   mydata(int data);
   mydata();
};
```

- In the above code, `mydata` class has three constructors having name `mydata`.
- One constructor is default with no parameter and other two constructors having string and integer parameter respectively.
- In the example, constructor name is same but the number of arguments and the type of argument is different.

2.10.2.2 References and Values

- **Java** by default uses reference. In **C++**, you can specify Reference by default it uses values.
- Each object is created explicitly with a reference model for variables so easier to check which constructor is called.
- This concept requires allocation from heap and extra indirections needed.
- Value model is more efficient to use but controlling initialization is very difficult.
- If variables are values then object created implicitly as a result of elaboration.
- The language allows objects to begin lifetime uninitialized, or allow to select an appropriate constructor for every elaborated object.
- Many object-oriented languages like **Java, Python, Ruby, Simula and Smalltalk** use a programming model where variables refer to objects.
- Languages such as **Ada95, C++, Modula-3 and Oberon** allows variable having a value which is an object.
- **Eiffel** by default uses a reference model but also allows the programmer which specifies certain classes should be expanded where case variables uses a value model.
- **C#** use structures to define types having variables are values and Class to define types having variables is references.
- **Ada 95, Modula-3 and Oberon** languages do not have constructors, elaborated objects begin life uninitialized and can use a variable before having a value.
- In **C++**, compiler ensures for every elaborated object an appropriate constructor is called, but rules used to identify constructors and their arguments sometimes can be confusing.

- If C++, variable of class type foo is declared without initial value then the compiler will call foo's zero-argument constructor

```
foo Q;              // calls foo::foo()
```

- If wants to call a different constructor, specify constructor arguments in declaration for overloading.

```
foo Q(100, 'A');// calls foo::foo(int, char)
```

- Argument list consists of a single object of the same or different class:

```
foo P;
bar Q;
….
foo c(P);     // calls foo::foo(foo&)
foo d(Q);     // calls foo::foo(bar&)
```

- Programmer's intent is declaring a new object having initial value same as the existing object. More natural to write:

```
foo P;        // calls foo::foo()
bar Q;        // calls bar::bar()
...
foo R=P;      //calls foo::foo(foo&)
foo S=Q;      //calls foo::foo(bar&)
```

- In C++, copy constructor is a single-argument constructor.
- In declaration, equals sign "="indicates initialization not an assignment.
- The effect is not the same as that of the similar code fragment.

```
foo P, R, S; // calls foo::foo() three times
bar Q;       // calls bar::bar()
...
R = P;       // calls foo::operator=(foo&)
S = Q;       // calls foo::operator=(bar&)
```

- R and S are initialized with the zero-argument constructor and equals sign indicates assignment not initialization.

2.10.2.3 Execution Order

- While creating a derived class object in C++, the compiler ensures that the base class' constructors are executed first, before the derived class' constructor.
- If a class has members which are objects of some class then the member's constructors will be called before the object's constructor in which they are contained.
- Rules are source of semantic and syntactic complexity: When combined with multiple constructors, built-in objects, and multiple inheritances, the result is a complex series of nested constructor calls that resolve overloads before control enters in the given scope.
- Other languages have simpler rules.

Rules for C++:

- C++ insists on every object be initialized before use.
- If the object's class (call it B) is derived from some other class (call it A), C++ insists on calling an A constructor before calling a B constructor, the derived class is guaranteed that inherited fields are never in inconsistent state.
- When an object of class B created (via declaration or a call to new), the creation operation specifies B constructor's arguments.
- When multiple constructors exist, C++ compiler resolves overloading with the help of these arguments.
- In C++, the header of the derived class's constructor specifies base class constructor arguments:

```
foo::foo(foo params):bar(bar args)
{
    ...
}
```

- foo is derived from bar in the above code.
- The list foo params consists of foo constructor's formal parameters.
- Between the parameter list and the opening brace of the subroutine definition is a "call" to a base class's constructor (class bar).
- The bar constructor's arguments can be arbitrarily complicated expressions involving the foo parameters.
- Before starting foo constructor's execution, compiler will arrange execution of bar constructor.
- If class B is derived class of class A, then constructor of A is called before constructor of B.
- To get constructor arguments, we should use an initialization list,

```
class B:A
{
    ...
}
B::B(B parameters):A(A parameters)
{
    ...
}
```

- The part after the colon is a call constructor of A.

Rules for Java:

- In Java, *super* keyword refers to the base class of the class in which it appears.
- If no call to super then Java compiler inserts a call to the base class's zero-argument constructor automatically (constructor must exist).

- For all objects Java uses reference model uniformly, any class members which are objects will be references than "expanded" objects.
- Initialized to null such as members.
- If wants something different, must call new explicitly within surrounding class's constructor.

Rules for C#, Eiffel and Smalltalk:

- **C#, Eiffel and Smalltalk** adopt a similar approach.
- In C#, struct type members are initialized by setting their fields to zero or null.
- In **Eiffel**, if a class contains expanded class type members, requires a single constructor with no arguments.
- When the surrounding object is created, the Eiffel compiler arranges to call this constructor.
- Initialization of base classes in Smalltalk, Eiffel, and CLOS are all more lax than C++.
- For each newly created object, compiler or interpreter arranges to call the constructor (creator, initializer) automatically.
- For base classes does not arrange to call constructors automatically, initializes default (zero or null) values to base class data members.
- Explicitly derived class's constructor(s) must call a base's constructor, if the derived class wants different behavior.
- In **Objective-C**, no special notion of constructor. Must write and invoke own initialization methods explicitly.

2.10.2.4 Garbage Collection

- Destructors are comparatively rare.
- In languages like C++, most common use is manual storage recovery.
- The need for a destructor is much less if the garbage collection is done automatically by the language implementation.
- When an object of any class is destroyed, the destructor of derived class is called first, and then the base class's destructors are called.
- Order of destruction is reverse the order of constructor.
- Purpose of destruction is to free allocated space from the heap.
- Languages such as Java, Smalltalk, Eiffel, etc. provide automatic garbage collection.

Java's finalize() method:

- In Java, the finalize() method can be overridden for explicit garbage collection.
- It allows code execution when the object is going to delete. But actually should not extend the lifetime of object by doing this as the `finalize()` method is called only single time per object.

Garbage collection in C++:

- In C++, when object is destroyed, derived class's destructor is called first, followed by base class(es) in derivation's reverse order.

- For example, consider the following code that create a list or queue of character-string names:

```
Class name_list: public gp_list
{
    char *name;          // pointer to the data in a node
    public:
    name_list()
    {
        name = 0;        // empty string
    }
    name_list(char *s)
    {
        name = new char[strlen(s)+1];
        strcpy(name, s);    // copy argument into member
    }
    ~name_list              //destructor
    {
        if (name != 0)
        {
            delete[] name; // reclaim space
        }
    }
};
```

- Destructor in class retrieves space allocated in the heap by the constructor.
- Less need for destructors in languages having automatic garbage collection.
- The concept of "destruction" is suspicious in a garbage collection language because the programmer has no control over the time at which the object will be destroyed.

Garbage collection in C# and Java:

- In C# and Java, the programmer declares a finalize method called immediately before reclaiming the space for an object by the garbage collector, but feature is not used widely.

2.11 DYNAMIC METHOD BINDING

2.11.1 Basic Concept

- A principal consequence of inheritance/type extension is: Derived Class D has all the data members and subroutines of its base class C.
- When class D does not hide publicly visible members of C, it allows class D's object to be used in any context which expects Class C's object.

- In Ada, a derived class that does not hide publicly visible members of its base class is a **subtype** of that base class.
- **Subtype Polymorphism:** It is ability to use a derived class in a context which expects its base class.
- Consider an administrative computing system for a base class person of bank, we might derive classes customer and employee from class person:

```
class person
{...
class customer:public person
{...
class employee:public person
{...
```

- Both `customer` and `employee` objects contains all the properties of a `person` object, should be able to use them in a person context:

```
customer c;
employee e;
...
person *p = &c;
person *q = &e;
```

- A subroutine like,

```
void person::print_mail()
{...
```

would be polymorphic- capable of accepting arguments of multiple types:

```
c.print_mail();      // i.e., print_mail(c)
e.print_mail();      // i.e., print_mail(e)
```

- With other forms of polymorphism, depends on the fact that `print_mail` uses those features of its formal parameter that actual parameters will have in common.
- But now let's say we have changed the definition of print_mail in both of these derived classes. For instance, let's say we want to write some data in the corner of this label.
- Now have multiple versions of subroutine: customer::print_mail and employee::print_mail rather than the single, polymorphic person::print_mail.
- Which version we will get depend on the object?

```
c.print_mail();      // customer::print_mail(c)
e.print_mail();      // employee::print_mail(e)
```

But what about,

```
x->print_mail();     // ??
y->print_mail();     // ??
```


- Does the choice of the called method depends on the variables types x and y, or on the objects classes c and e to which those variables refer?
- The first (use reference type) is known as **static method binding.**
- The second (use object's class) is known as **dynamic method binding.**
- For Object-oriented programming, dynamic method binding is central.
- Binding can be done when calls to virtual methods are given at run time based on the object's class.
- **Simula:** Virtual methods are listed at beginning of the class declaration.

```
class a;
virtual: procedure vp;
begin
   ...
end a;
```

- C++: Keyword "virtual" prefixed to declaration of function.

```
class person
{
   public:
   virtual void VP ();
   ...
}
```

- This means that you need to store a virtual table with every single object.
- Overriding derived class B will override base class function named `display`.

Example:

```
class A
{
   virtual void display()
   {
      System.out.println("hi in A");
   }
}
class B extends A
{
   void display()
   {
      System.out.println("hi in B");
   }
```

```
class C
{
    public static void main ()
    {
        B b=new B();
        b.display(); // hi in B
    }
}
}
```

Dynamic vs. Static Binding:

- **Static method binding** uses reference so can be shown as,

```
s.methodname();
p.methodname();
```

- **Dynamic method binding** uses object class so used as pointer.

```
x->methodname();
y->methodname();
```

Example: Java uses dynamic binding as,

```
public class A
{
    public String toString()
    {
        return "dynamic string";
    }
    public static void main (String args[])
    {
     Object obj = new A();
     System.out.println(obj);
    }
}
```

Semantics and Performance:

- Dynamic method binding enforces run-time overhead. The overhead is low, but it is still a concern for smaller sub-routines in high-performance environments.
- For all methods, **Modula-3, Objective-C, Python, Ruby and Small-talk** uses dynamic method binding.
- By default **Java and Eiffel** uses dynamic method binding, individual methods and classes are labeled final(Java) or frozen(Eiffel), so that cannot be overridden by derived classes and can employ an optimized implementation.

- **Ada 95, C++, C# and Simula** uses static method binding by default, specifies dynamic binding when desired.
- Common terminology in these languages is to differentiate between method overriding, which is based on dynamic binding, and methods redefining which is based on static binding.
- C# requires explicit use of the keywords *override* and *new* whenever an override method in a derived class or method redefines (respectively) of the same name in a base class.

2.11.2 Virtual and Non-Virtual Methods

- In **C++, C# and Simula** by default uses static method binding.
- By labeling particular methods as virtual, the programmer specifies those methods use dynamic binding.
- Virtual methods calls are dispatched to the appropriate implementation at run time based on object's the class rather than the reference type.

Examples:

- In **C++ and C#**, the keyword *virtual* prefixes the subroutine declaration:

```
class person
{
    public:
    virtual void print_mail();
    ...
}
```

- In **Simula**, virtual methods are listed at the class declaration's beginning:

```
CLASS Person;
VIRTUAL:PROCEDURE PrintMail;
BEGIN
...
PROCEDURE PrintMail...
COMMENT body of subroutine
...
END Person;
```

- In **Ada 95**, programmer associates it with certain references rather than associating dynamic dispatch with particular methods.

2.11.3 Abstract Classes

Abstract Class:

- A class having at least one abstract method regardless of declaration syntax.
- If class has one or more abstract methods then class becomes abstract.

- It is not possible to declare an abstract class's object because missing at least one member function definition. This means that you cannot create an object for an abstract class.

Purpose of an Abstract class:

- Abstract class serves as a base for the concrete class.
- Concrete class should contain a definition for each inherited abstract method which inherited by it.

Application to Dynamic Method Binding:

- This allows the code that calls methods a base class objects by considering the concrete methods will be invoked at elaboration time.
- Consider the code below,

```
abstract class A
{
   abstract void display();
   void f() {
   System.out.println("hello abstract class A");
}
class B extends A
{
   void display(){
   System.out.println ("hi derived class B");
}
}
```

- In a base class, an abstract method's existence provides a "hook" for dynamic method binding.
- It allows the programmer to write code which calls object's methods of the base class assuming at run time appropriate concrete methods will be invoked.

Abstract Methods:

- Virtual methods without definition or body called Abstract Methods.
- **In C++:** It is called pure virtual method, where procedure declaration has assignment with zero.

```
class person
{
   ...
   public:
   virtual void print_mail()=0;
   ...
}
```


- Consider code below,

```
class A
{
   ...
   public:
   virtual void myvirtu()=0;
}
```

- In most object-oriented languages possible to omit the body of a virtual method in a base class.
- **In Java and C#:**It can be done by labeling both the class and the missing method as abstract:

```
abstract class person
{
   ...
   public abstract void print_mail();
   ...
}
```

- All virtual methods are abstract in **Simula**.

Interface:

- Interface is pure abstract class which has all abstract methods.
- **In Ada 2005, C#, and Java Interfaces:** Classes having no members other than abstract methods, any fields or method bodies.
- Supports restricted, "mix-in" form of multiple inheritance.

```
interface A
{
   void display();
   void f();
}
class B implements A
{
   void display()
   {
      System.out.println ("hi Smaran");
   }
   void f(){}
}
```

2.11.4 Member Lookup (vtable)

- With static method binding like in Ada 95, C++, C#, or Simula, compiler always tell method's version to call based on the variable's type being used.
- Every object is represented with a record whose first field contains virtual method table's address (vtable) for that object's class, in dynamic binding.
- Our objects becomes more complicated for the compiler to handle:
  - Virtual method tables
  - Reference counts
- Consider the code below,

```
class foo
{
    int a;
    double b;
    char c;
    public:
    virtual void k (...)
    virtual int l (...)
    virtual void m();
    virtual double n( (...)
    ...
} F;
```

- vtable contains all virtual methods of foo class. It can be shown as,

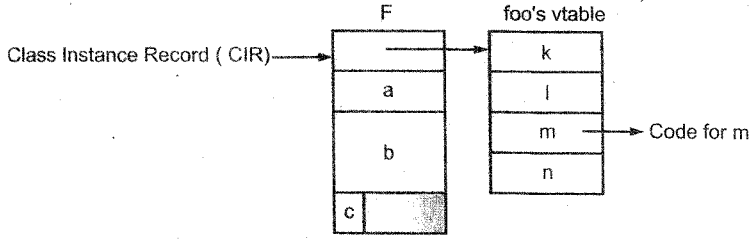


Fig. 2.15: Member lookup or vtable

- In dynamic method binding, the object referred to by a reference or pointer variable must contain sufficient information to compiler's code generated to find the method's right version at run time.
- **vtable**: It is an array whose i^{th} entry indicates the address of the code for the object's i^{th} virtual method.
- A given class's all objects share the same vtable. Suppose, for methods the `this(self)` pointer is passed in `register r1`, m is the third method of class. Foo, and F is a pointer to class foo's object.

- Code to `call f→m()` will be as follows:

```
r1:= f
r2:=*r1                  -- vtable address
r2:=*(r2 + (3-1) × 4)    -- assuming 4 = sizeof (address)
call*r2
```

- Extra overhead can be avoided when compiler deduce the relevant object type at compile time. The deduction is trivial for calls to object-valued variables methods.
- If bar is derived from foo, place its additional fields at the end of the "record" which represents it.
- By copying the vtable for foo create a vtable for bar, replacing the entries of virtual methods overridden by bar, appending entries for any virtual methods declared in bar.
- If having class bar's object, can assign its address into a variable of type foo*:

```
class bar: public foo
{
    int w;
    public:
    void m(); //override
    virtualdouble s(...
    virtual char *t(...
    ...
} B;
```

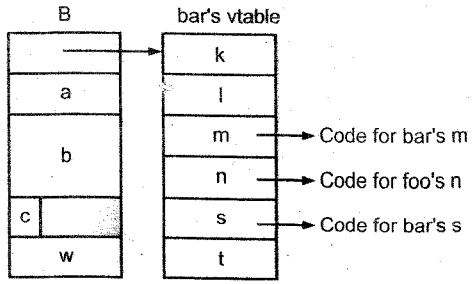


Fig. 2.16: Implementation of Single Inheritance

- In above figure, there presentation of object B begins with its class's vtable address. First four entries represent same members as for foo, except one has been overridden, now contains the address of the code for a different subroutine. Additional fields of bar follow the ones inherited from foo in B's representation; additional virtual methods follow the ones inherited from foo in class bar's vtable.

```
class foo
{...
class bar: public foo
{  ...
   ...
   foo F;
   bar B;
   foo* q;
   bar* s;
   ...
   q = &B;   // ok; will be used prefixes of B's data space
             and vtable, references through q
   s = &F;   // static semantic error; F lacks the additional
             // data and vtable entries of a bar
}
}
```

- In C++, statically compiler verifies the type correctness of this code.
- We doesn't know which Object's class it will refer to at runtime, but it knows either foo or something derived (directly or indirectly) from foo will be there,
- Ensure that all the members can be accessed using the foo-specific code.
- In C++, "backward" assignments by means of a `dynamic_cast` operator is allowed: `s = dynamic_cast<bar*>(q);// performs a run-time check`
- Null pointer is assigned to s when the run-time check fails.
- **C++** supports traditional C-style casts of object pointers and references for backward compatibility:

```
s = (bar*)q;          // permitted, but risky
```

- With a C-style cast, programmer ensures the actual object involved is of an appropriate type: a dynamic semantic check is not performed.
- **C# and Java** uses traditional cast notation, but performs the dynamic check.
- In **Eiffel**, a reverse assignment operator, ?= assigns an object reference to a variable only if the type at run time is acceptable:

Example:

```
class foo...
class bar inherit foo...
...
f: foo
b: bar
...
f:= b      -- always ok
b ?= f     -- reverse assignment: b gets f if f refers to a bar
           object at run time; otherwise b gets void
```


- In **C#**, an as operator performs a similar function.
- Variables are untyped references in **Smalltalk**.
- A reference to object may be assigned into variable.
- When code actually tries to call an operation at execution time, the language implementation checks if the operation is implemented by the object.
- The implementation is straightforward. Object's fields are never public; the only means of object interaction provided by methods.
- Representation of an object begins with a type descriptor's address. The type descriptor has a dictionary which maps method names to code fragments.
- At run time, the Smalltalk interpreter performs a lookup operation in dictionary to check the method is supported or not. If not, generates a "message not understood" error.
- In CLOS, in Objective-C and in the Object-oriented scripting languages, there are similar semantics that call for similar implementations.
- The dynamic approach is more flexible than the static, but incurs significant run-time cost and delays the reporting of errors.
- Imposing the overhead of indirection as well as at compile time, virtual methods prevent in-line expansion of subroutines.
- When subroutines are small and frequently called, lack of in-line subroutines can be a serious performance problem.
- C, C++ try to avoid run-time overhead hence use of static method binding as the default and there is a lot of dependency on object-valued variables for which virtual methods can be dispatched at compile time.

2.11.5 Polymorphism

- In Polymorphism,
  - A derived class B has all the base class A members.
  - Class B can be used anytime when class A is expected.
  - If any publicly visible members not hidden of class Ain class B, then class B is a subtype of class A.
  - If class B is used in place of class A, this is a form of polymorphism.

Example:

```
class A { …}
class B: public A { …}
class C: public A { …}
B b;
C c;
…
A *x = &b;
A *y = &c;
```

- Dynamic method binding introduces Polymorphism (subtype polymorphism specifically) into any code expecting a reference to some base class's (foo) object.
- So long as derived class's object support the base class's operations, code will work equally well with references to any class's objects derived from foo.
- Declare a reference parameter of class 'foo'. For instance, if the programmer declares subroutine that will only use 'foo features' and will execute on the object that provides those features.
- One might think that the combination of Inheritance and Dynamic Method Binding would eliminate the need for generics, but this is not the case.
- For example, in the gp_list_node class and its descendants. By placing structural aspects of an abstraction in a base class, make it easy to create type-specific lists: int_list_node, float_list_node, student_list_node, etc.
- Base class methods like predecessor and success or return base class type's references which do not support type-specific operations.
- Must perform an explicit type cast for accessing the values stored in objects returned by the list manipulation routines:

```
int_list_node_ptr q, r;
...
r = q->successor();     // error: type clash
gp_list_node_ptr p = q->successor();
cout<<p.val;        // error: gp_list_nodeshave no value
r = (int_list_node_ptr) q->successor();
cout<<r.val;        // ok
```

- Here, the type cast on next to last line is awkward and unsafe.
- We can't use a dynamic_cast operation as gp_list_node has no virtual members and hence no vtable.
- We can simply redefine the methods as follows:

```
int_list_node* int_list_node::predecessor()
{       // redefine
return(int_list_node*) gp_list_node::predecessor();
}

int_list_node* int_list_node::successor()
{       // redefine
return(int_list_node*) gp_list_node::successor();
}
```

- Redefining all the appropriate arguments and base class method's return types in every derived class is tedious exercise, and the code is still unsafe.
- Compiler cannot verify type correctness.

- Generics get around both problems.
- For purpose of abstracting over unrelated types generics exist, inheritance does not support, in a nutshell.
- **C#, Eiffel and Java** provide generics.
- **Java**'s version is simpler than the others: As object variables are references always have the same size always and a single copy of the code can be shared by every instance of a generic generally.
- **Eiffel** allows the programmer to declare parameters and return method's values of the same type as class's some "anchor" field, as convenient shorthand.
- If a derived class redefines the anchor, parameters and return values are automatically redefined without specifying them explicitly as follows :

```
Class gp_list_node...
   ...
Class gp_list
feature {NONE}          -- private
header: gp_list_node    -- to be redefined by derived classes
feature {ALL}           -- public
head: like header is... -- methods
append(new_node:like header) is...
   ...
end
   ...
Class student_list_node inherit gp_list_node
   ...
Class student_list
inherit gp_list
redefine header end
feature {NONE}
header:student_list_node
   --don't need to redefine head and append
end
```

- The like mechanism which does not eliminate need for generics, but makes easier to define them.

2.11.6 Object Closures

- Object closures used in an object-oriented language to achieve the same effect as subroutine closures with nested subroutines, which encapsulate a method with context for later execution.
- This mechanism relies for full generality on dynamic method binding.

- Recall the plus_x object closure, apply_to_A rewritten in generic form:

```
template<class TP>
class un_op
{
public:
   virtual TP operator()(TP i) const = 0;
};
class plus_x: public un_op<int>
{
const int x;
public:
   plus_x(intnum): x(num)
   { }
virtual int operator()(int i) const
{
   return i+x;
}
};
void apply_to_A(const un_op < int>&f, int A[], int A_size)
{
   int i;
   for (i = 0; i< A_size; i++) A[i] = f(A[i]);
}
...
int A[10];
apply_to_A(plus_x(2), A, 10);
```

- In the above code, an object derived from un_op<int> passed to apply_to_A.
- The "right" function will be called as operator()is virtual.
- Useful idiom for many applications is to encapsulate a method and its arguments in an object closure for later execution.
- Suppose, writing a discrete event simulation.
- Might like a general mechanism, schedules a call to an arbitrary subroutine with an arbitrary set of parameters to occur at future point in time.
- If the subroutines want to have called varies in numbers and types of parameters, won't be able to pass them to a general-purpose schedule_at routine.
- We can solve this problem with object closure.

- This same technique is used in Java, For example,

```
Class fn_call
{
public:
   virtual void trigger() = 0;
};
Void schedule_at(fn_call &fc, time t)
{
   ...
}
   ...
void foo(int x, double y, char z)
{
   ...
}
Class call_foo:public fn_call
{
int arg1;
double arg2;
char arg3;
public:
   call_foo(int x, double y, char z): // constructor
   arg1(x), arg2(y), arg3(z)
   {
      // member initialization is all that is required
   }
void trigger()
{
   foo(arg1, arg2, arg3);
}
};
   ...
call_foocf(3, 3.14, 'p'); // declaration/constructor call
schedule_at(cf, now() + delay);
   // at some point in the future, the discrete event system will
   // call cf.trigger(), will cause a call to foo(3, 3.14, 'p')
```

- Create new control threads to encapsulate initial arguments. You can implement iterators using the visitor pattern.

2.12 MULTIPLE INHERITANCE

- Before to learn multiple inheritance, let us see what is Single Inheritance.
- Consider the code below:

```
Class bar:public foo {
   int w;
public:
   void m(); //override
   virtualdouble s(...
   virtual char *t (...
   ...
}B;
```

- Here, class bar is inherited from class foo which is declared in above code snippet.
- Class bar also have its own virtual methods so these also added to existing vtable, which can be shown as,

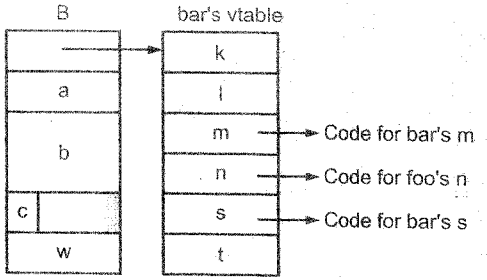


Fig. 2.17: **Vtable for class foo and bar**

2.12.1 Introduction

- Multiple inheritance means one class derived from two or more base classes. A derived class inherits features from more than one base class.

Example, in C++:

```
Class student:public person,public gp_list_node
{ … }
```

- Class student's object will have all the fields and methods of both a person and a gp_list_node.

Example, in Eiffel:

```
class student
inherit
        person
        gp_list_node
feature
...
```


- Supported in C++, Eiffel, CLOS and Python.
- There is Single Inheritance in the following languages: Ada 95, Modula-3, Oberon, Objective-C, Simula, Smalltalk.
- C#, Java and Ruby provides limited support "mix-in" form of multiple inheritance.
- One parent class is permitted to have fields.

Reasons to use Multiple Inheritance:

- It involves a number of adjustments.
- How do you decide?
  - Whether it satisfies "is a" relationship?
  - Is creation of object speed a constraint?

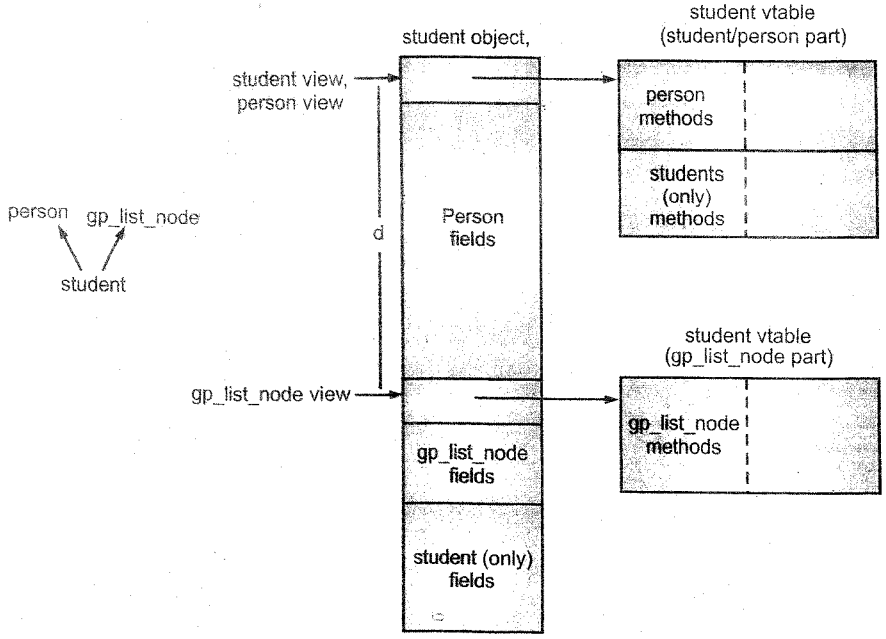


Fig. 2.18: **Compile-time Allocation and v-table**

Types of Multiple Inheritance:

- There are different types of multiple inheritance:
  - Normal (non-repeated)
  - Shared
  - Mix-in
  - Repeated

2.12.2 Shared Inheritance

- Repeated inheritance having a single copy of the grandparent.
- Shared inheritance allows sharing of common parent classes. It is type by default in Eiffel. Example, Person class in the above tree structure.

- This inheritance still have problem when inheriting methods that overridden.
- Consider class hierarchy,

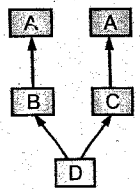


Fig. 2.19: Class Hierarchy

- Its allocation and vtable can be shown as,

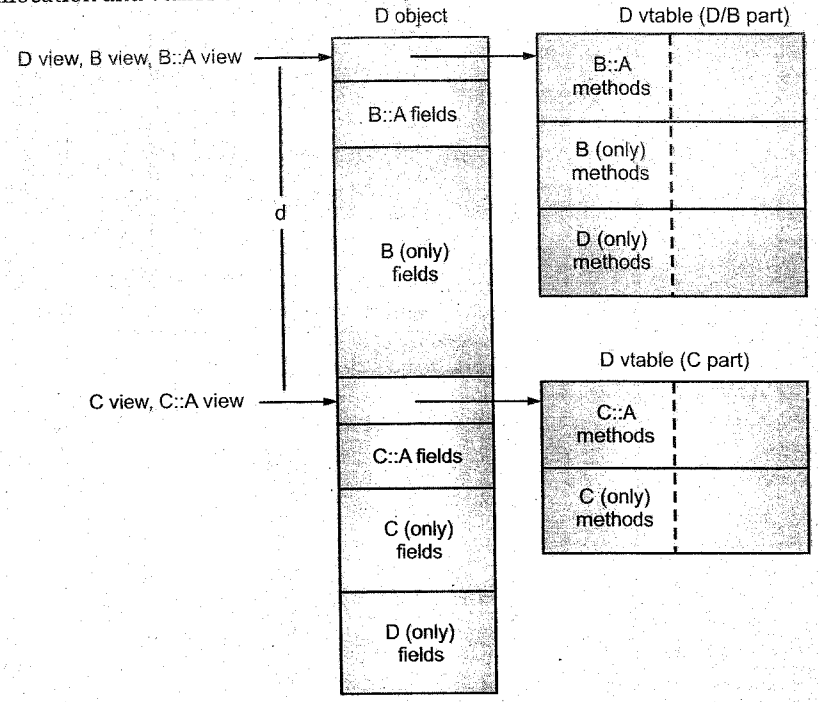


Fig. 2.20: Allocation and vtable of Shared Inheritance

- In C++, replicated inheritance is default but shared inheritance is possible.
- Consider the following code:

```
class A {... };
class B: public virtual A { … };
class C: public virtual A { … };
class D: public B, public C { … };
```


- Creates the following scenario:

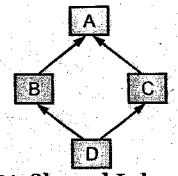


Fig. 2.21: Shared Inheritance

2.12.3 Mix-In (MI) Inheritance

Mix-In Inheritance:

- Inheritance from one "real" base class and number of interfaces.
- In Mix-In Inheritance, only one of the base classes may contain method definitions.
- There are only abstract methods in other type inheritance and the base class(es).
- Only type of MI inheritance supported in Java.
- In Java, inheritance can be achieved using keyword *extends*.
- Mix-in (interface) inheritance in Java can be achieved using keyword *implements*.

Java Interfaces :

- Consider the following example:
  `public class String extends Object implements Serializable, CharSequence, Comparable.`
- Here, class is extended from Object and interfaces implemented such as Serializable, CharSequence, Comparable.
- Java interfaces can contain definition prototypes and static variables.
- In C++, Mix - in interface can be achieved by making the interfaces be abstract base classes and the base class's every method declared as pure virtual.

2.12.4 Semantic Ambiguities

- **What if two base classes have implementations of a shared method?**
  - In Eiffel or C++, it will not work.
  - In some languages, calling methods happens explicitly, i.e. class::method()
- **What if the relationship below occurs?**
- This is repeated multiple inheritance.
  - As one of the base class /ancestors is repeated in the parent class of one of the derived class / descendants.

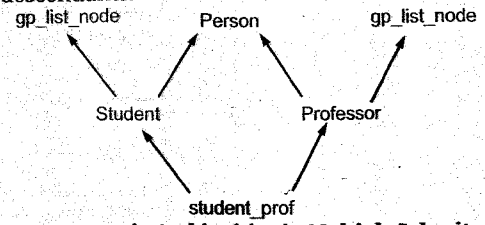


Fig. 2.22: Semantic Ambiguities in Multiple Inheritances

2.12.5 Replicated Inheritance

- **Repeated Inheritance:** It is a multiple inheritance having common "grandparent".
- **Replicated Inheritance:** It is a repeated inheritance having separate copies of the grandparent.

- Even if both derived from the same grandparent, replicated inheritance does not share same common grandparent.
- It is default type in C++.
- Example, `gp_list_node` in the above tree structure can only directly access one level deep.To access student members of `gp_list_node`, you must first assign a `student_prof` pointer into a student or professor pointer.
- Consider the class hierarchy,

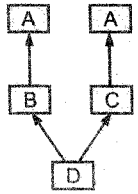


Fig. 2.23: Class Hierarchy

- Its allocation and vtable can be shown as,

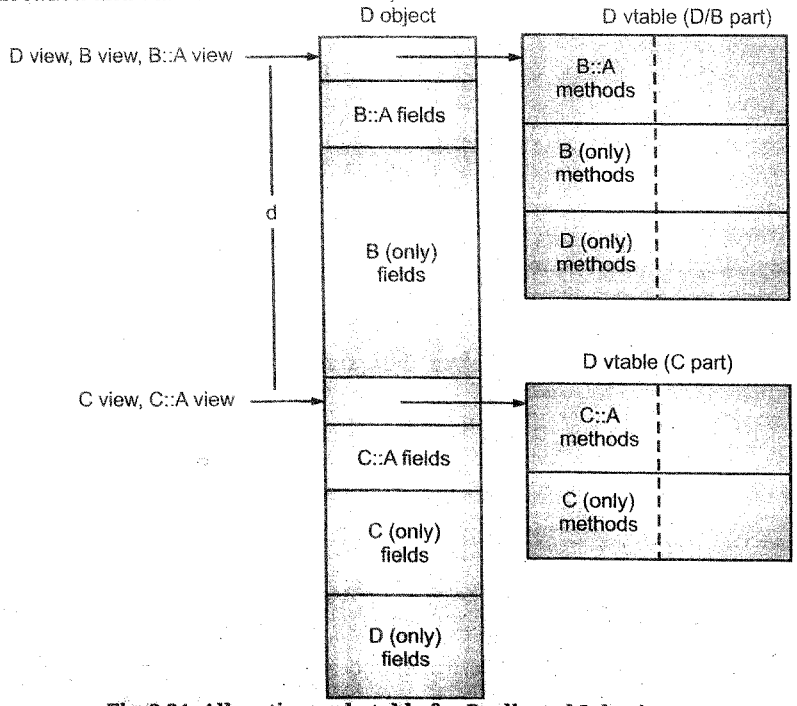


Fig. 2.24: Allocation and vtable for Replicated Inheritance

- In C++, consider the following code:

```
class A { … };
class B: public A { … };
class C: public A { … };  /* Derives from A but it is not shared! */
class D: public B, public C { … };
…
```


- Given the following situation:

```
A* a; B* b; C* c; D* d;
a = d;    //error ambiguous
b = d;    // ok
c = d;    // ok
a = b;    // ok
a = c;    // ok
```

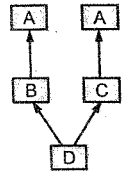


Fig. 2.25: Replicated Inheritance

SUMMARY

- Using object closure, we can encapsulate our subroutine as a method of a simple object.
- When *restrict* keyword attached to a pointer declaration we can not make aliases of it.
- Explicit parametric polymorphism is also known as Generosity.
- Coercion is the process by which a compiler automatically converts a value of one type to another means it is implicit type casting.
- Types of polymorphism are Parametric, Subtype and Ad-hoc Polymorphism.
- Elaboration is the process by which declaration becomes active.
- At compilation, Symbol Table maintains dimension and bounds information of each array.
- Dope Vectors (descriptor) is the compiler generate code in a dope vector at run time.
- One of the alternatives for static chain is to use a display, which is array to store the static links.
- In a C++, Java programming language, Ad-hoc polymorphism achieved with a function overloading.
- Three fundamental concepts of object-oriented programming: Encapsulation, Inheritance and Dynamic Method Binding.
- Encapsulation allows the complicated data abstraction's details to be hidden behind a comparatively simple interface.
- For programmers, Inheritance extends utility of encapsulation by making it easy to define new abstractions as refinements or extensions of existing abstractions.
- For polymorphic subroutines, inheritance provides a natural basis: if a subroutine expects a given class's instance as argument, then any class's object derived from the expected one can be used instead.

- Dynamic method binding extends this kind of polymorphism by arranging for a call to one of the parameter's methods to use the implementation associated with the actual object's class at runtime, rather than the implementation associated with the parameter's declared class.
- Languages like Ada 95, FORTRAN 2003, Modula-3 and Oberon support object orientation through a type extension mechanism in which encapsulation is associated with modules and inheritance and dynamic method binding are associated with a special form of record.
- Treating variables as references rather than values leads to simpler semantics, requires extra indirection.
- Garbage collection eases the creation and maintenance of software, but imposes run-time costs.
- In Dynamic Method Binding, methods are dispatched using vtables or some other lookup mechanism.
- Simple implementations of multiple inheritance impose overheads even when unused.
- In-line subroutines improves the performance of code with many small subroutines, by eliminating the overhead of the subroutine calls themselves, by allowing register allocation, common sub-expression analysis, and other "global" code improvements to be applied across calls.
- At the same time, in-line expansion generally increases the size of object code.
- Despite lack of multiple inheritance, Smalltalk is purest and most flexible object-oriented languages.
- Its lack of compile-time type checking with its "message-based" model of computation and need for dynamic method lookup, render implementations rather slow.
- C++ with object-valued variables, default static binding, minimal dynamic checks and high-quality compilers largely responsible for the growing popularity of object oriented programming.
- Improvements in reliability, maintainability, and code reuse may or may not justify the high performance overhead of Smalltalk.
- Almost certainly justify the relatively modest overhead of C++, and slightly higher overhead of Eiffel.
- With increasing size of software systems, the explosive growth of distributed computing on the Internet and the development of highly portable Object-oriented languages (Java), Object-oriented scripting languages (Python, Ruby, PHP, JavaScript), and Binary object standards, Object oriented programming will clearly play a central role in the 21st century computing.

PRACTICE QUESTIONS

Q.I Multiple Choice Questions.

1. Modularity can be achieved using ____.
   (a) Encapsulation (b) Interfaces
   (c) Separate Compilation (d) All of the mentioned
2. The Basis of encapsulation is _____.
   (a) Class (b) Object
   (c) Member function (d) All of the above
3. Generic data structure can be implemented using _____.
   (a) Function Template (b) Class Template
   (c) File Template (d) Inheritance
4. _____ is possible to declare as Friend.
   (a) Member function (b) global function
   (c) Class (d) All of these
5. Recursion works with the help of _____ data structure.
   (a) Array (b) List
   (c) Queue (d) Stack
6. A Stack frame or Activation record in recursion consist of _____.
   (a) Parameters to be processed by the called function
   (b) Local variables in the calling function
   (c) The return address
   (d) All of the above
7. In recursive call the return address of called function is stored on _____.
   (a) Stack frame (b) Queue
   (c) Tree (d) Cache
8. Which of the following statement is correct?
   (a) A constructor is called at the time of declaration of an object.
   (b) A constructor is called at the time of use of an object.
   (c) A constructor is called at the time of declaration of a class.
   (d) A constructor is called at the time of use of a class.
9. The situation when in a linked implementation of stack overflow occurs, we cannot perform _____.
   (a) PUSH operation (b) POP operation
   (c) Both (a) and (b) (d) None of the above

10. A data structure whose size is determined at compile time and cannot be changed at run time is _____.
    (a) ADT (b) static
    (c) ephemeral (d) permanent
11. Which of the following concepts means adding new components to a program as it runs?
    (a) Data hiding (b) Dynamic binding
    (c) Dynamic loading (d) Type casting
12. The term _____ means the ability to take many forms.
    (a) Inheritance (b) Polymorphism
    (c) Member function (d) Encapsulation
13. Including only necessary details and ignoring additional details while defining a class is known as _____.
    (a) Overloading (b) Data abstraction
    (c) Polymorphism (d) Encapsulation
14. Which of the following concepts means waiting until runtime to determine which function to call?
    (a) Data hiding (b) Dynamic binding
    (c) Dynamic casting (d) Dynamic loading
15. Select the correct syntax of template _____.
    (a) Template (b) Template<>
    (c) Temp (d) None of these
16. Which of the C++ feature allows you to create classes that are dynamic for using data types?
    (a) Templates (b) Inheritance
    (c) Polymorphism (d) Information hiding
17. An object is _____.
    (a) Variable of class data type
    (b) Same as class
    (c) Just like global variable
    (d) Collection of data members and member functions
18. Which of the following concept of OOPs allows compiler to insert arguments in a function call if it is not specified?
    (a) Call by value (b) Call by reference
    (c) Default arguments (d) Call by pointer

19. Which is the longest scope in the following code?

```
#include<stdio.h>
intX;
int main()
{ intY; fun(); return 0; }
void fun(){ intZ; }
```

    (a) X (b) Y
    (c) Z (d) Both (a) and (b)
20. Select the correct statement?
    I . Procedure is a kind of routine that returns a value.
    II. Function is a kind of routine that does not return any value.
    (a) Only I (b) Only II
    (c) Both I and II (d) None of these

Answers

1. (d)	2. (d)	3. (c)	4. (d)	5. (d)	6. (d)	7. (a)	8. (a)	9. (a)	10. (b)
11. (c)	12. (b)	13. (b)	14. (b)	15. (b)	16. (a)	17. (a)	18. (c)	19. (a)	20. (d)

Q.II Answer the following questions in short.

1. What are the two potential problems with the Static Chain Methods?
2. What is display()?
3. Describe the deep access method of implementing dynamic scoping?
4. Describe the shallow access method of implementing dynamic scoping?
5. What is an association list?
6. What is a frame pointer? What is the use of it?
7. What is a calling sequence?
8. What is elaboration?
9. What is a referencing environment?
10. Explain the closest nested scope rule.
11. What is the purpose of a scope resolution operator?
12. What is a static chain? What is it used for?
13. What does it mean for a scope to be closed?
14. Explain the purpose of a compiler's symbol table.
15. Name three important benefits of abstraction.
16. What are the more common names for subroutine member and data member?
17. What is the purpose of the:: operator in C++?
18. What are constructors and destructors?
19. Write a short note on: (i) Inheritance, (ii) Abstraction.
20. What are private types in Ada?
21. Explain the significance of the *this* parameter in Object-oriented languages.

22. What are extension methods in C#? What purpose do they serve?
23. Explain Java inner class.
24. What is vtable? How to use it?
25. Define the Pure virtual function.

Q.III Answer the following questions.

1. Explain memory allocation (Explain 3 allocations in detail).
2. What is Stack base allocation?

   Hint:

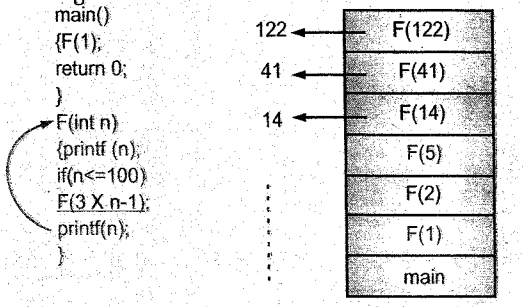
   (i) sp
   (ii) fp
   (iii) Temporaries
   (iv) Local variables
   (v) Book keeping info
   (vi) Return address
   (vii) Arguments and List
3. What is binding time?
4. Explain the distinction between decisions that are bound statically and those that are bound dynamically.
5. What is the advantage of binding things as early as possible?
6. What is the advantage of delaying or late bindings?
7. Explain the distinction between the lifetime of a name-to-object binding and its visibility.
8. What determines whether an object is allocated statically, on the stack, or in the heap?
9. List the objects and information commonly found in a stack frame.
10. What are internal and external fragmentations?
11. What is garbage collection?
12. What is a dangling reference?
13. What do we mean by the scope of a name-to-object binding?
14. Describe the difference between static and dynamic scoping.
15. What are forward references? Why are they prohibited or restricted in many programming languages?
16. Explain the difference between a declaration and a definition.
17. Trace the output:

```
main()
{
   F(1);
   return 0;
}
```



```
F(int n)
{
   printf (n);
   if(n<=100)
   F(3Xn-1);
   printf(n);
}
```

Hint: See the Tracing below:



1	2	5	14	41	122	122	41	14	5	2	1

Fig. 2.26

18. Using all 4 types of scope rules try to find output for following code,

```
intprint_base = 10;
print_integer (int n)
{
   switch (print_base)
   {
      case 10:...
      case 8:...
      case 16:...
      case 2:...
   }
}
foo
{
   print_integer (10);
   intprint_base = 16;
   print_integer (10);
   print_base = 2;
   print_integer (4);
}
```

19. Using all 4 types of scope rules try to find output for following code,

```
n : integer -- global declaration
procedure first
    n := 14.
procedure second
    n : integer -- local declaration
    first()
    n := 28.
if read integer() > 0
second()
else
first()
write integer(n)
```

20. Explain encapsulation.
21. Explain inheritance and its types in brief.
22. Differentiate between private, public and protected members in C++.
23. Differentiate between replicated and shared inheritance with proper example.
24. Explain any 2 concepts of object oriented programming in brief.
25. Show implementation of shared multiple inheritance ,when class hierarchy is shown as below:

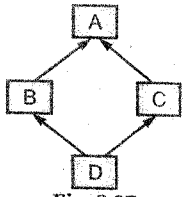


Fig. 2.27

26. Show descriptor of replicated multiple inheritance, when class hierarchy is shown as below:

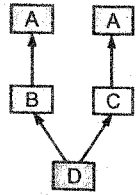


Fig. 2.28

27. Explain dynamic method binding.
28. Write a short note on: (i) Encapsulation, (ii) Polymorphism.
29. Show vtable allocation for the following code:

```
class A1 { … };
    class A2: public A1 { … };
    class A3: public A1 { … };
/* Derives from A1 but it is not shared! */
    class A4: public A2, public A3 { … };
```

■■■

CHAPTER 3

Data Types

Contents

Objectives ...

After learning this chapter you will be able:

- To get information about the concept of Data types and characteristics of common data types.
- To design Enumeration and subrange types.
- To design Issues and design choices made by designers.
- To learn implementation of various data types.
- To do the comparative study of primitive and user defined data types.
- To use pointers very efficiently avoiding problems of dangling reference.

3.1 INTRODUCTION

- In Computer programs, results are produced by manipulating data.
- Program may be viewed as a set of operations applied to specific data in a specific sequence.
- Types of data allowed, types of operation available and the mechanism provided for controlling the sequence operations applied to the data differs for different languages.
- ALGOL 68 provided a few basic types and flexible structure defining operators that allow a programmer to design a data structure for each need.

Data type:

- A collection of data objects and a set of predefined operations on those objects define a data type.

Descriptor:

- The collection of the attributes of a variable is called descriptor.
- In implementation, a descriptor is a collection of memory cells which stores variable attributes.
- For static attributes, descriptors are required only at compile time.
- These descriptors are built by the compiler as a part of the symbol table which used during compilation.
- For dynamic attributes, during execution part or all of the descriptor must be maintained.
- It is used for type checking as well as allocation and de-allocation operations are done.

Object:

- An instance of a user-defined (abstract data) type is represented by an object. Object associated with the value of a variable and the space occupied by it.

Rules for Data type:

- Data Type system consists of following rules:
  1. **Type equivalence rules** determine whether data type of two values is same.
     Example,

```
int a=2
int b=2
```

Types of Type equivalence: Type equivalence is categorized in following two types:

(i) A language in which aliases are considered distinct is called **Strict name equivalence.**

(ii) A language in which the aliases are considered equivalent is called **Loose name equivalence.**

  2. **Type compatibility rules** determine when value of a given type can be used in a given context.
     Example,:

```
int X=5
String Y="5"
```

3. **Type inference rules** define the type of expression based on its result.
   Example:
   ```
   int c=2+4.6
   ```

- **Type checking:** This is a process of ensuring that a program follows the language's type compatibility rules. A violation of any of these rules is known **a Type clash.**
- **Type conversion** : This process refers to change a value of one data type into another.
- **Type coercion**: Whenever a language allows a value of one type to be converted to another, by implicit conversion to the expected type. This conversion is called as **Type coercion.**

3.2 Primitive Data Types

- More or less all programming languages provide a set of primitive data types.
- **Primitive data types** are the data types which are not defined in terms of other data types.
- Some primitive data types are only reflections of the hardware, for example integer types.
- Others data types require only a little non-hardware support for implementation.
- The primitive data types along with one or more type constructors provide structured types.

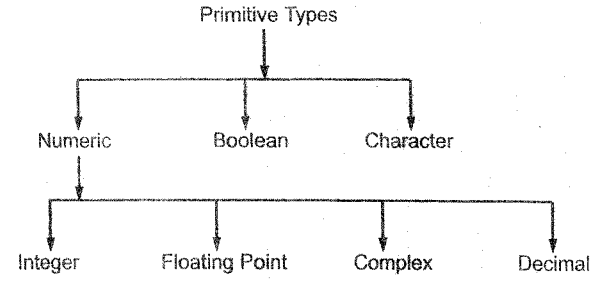


Fig. 3.1: Classification of Primitive Data Types

3.2.1 Numeric Types

- Early programming languages had numeric primitive types. These play central roles among types supported by modern languages.
- We can have different numeric data types such as integer, float and so on.
- We can perform arithmetic operations like addition and subtraction, bit operations, comparison, assignment with all numeric types.

3.2.1.1 Integer

- Integer is the most common primitive numeric data type.
- The computers support several sizes of integer and capabilities are reflected in programming languages.
- For example, Ada supports three integer types, namely SHORT INTEGER, INTEGER, LONG INTEGER.

- Java supports signed integer sizes as: BYTE, SHORT, INT, LONG.
- C++ and C# supports unsigned integer types i.e. integer values without signs.
- In computer, value is represented by string of bits.
- The sign is represented by the leftmost bit, either it is positive or negative.
- It supported directly by the hardware so the mapping is trivial.
- For negative integer sign bit is set, remainder of the bit string represents the absolute value of the number.
- Two's compliment is used to store negative integers.

3.2.1.2 Floating Point

- Floating point numbers are model real numbers as approximations only.
- In computers, Floating point numbers are stored in binary.
- Arithmetic operations results into loss of accuracy.
- The representation of values is in the form of fractions and exponents.
- Languages for scientific purpose supports minimum two floating-point types (e.g. float, double), sometimes more.
- Float type is the standard size that stored in four bytes memory.
- When larger fractional parts are needed, the Double data type is used. It occupies twice memory as that of Float. It also provides twice the number of bits of fraction.
- According to IEEE, Single precision floating point format is as follows:

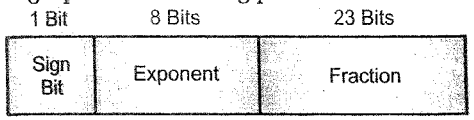


Fig. 3.2 (a): Single precision Floating point Format

- These numbers can be represented from 0 to 31 left to right.
- According to IEEE, Double precision floating point format is as follows:

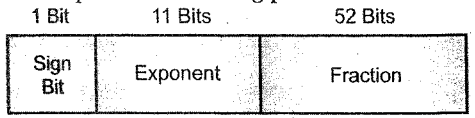


Fig. 3.2 (b): Double precision Floating point Format

- These numbers can be represented from 0 to 63 left to right.
- Floating point types is defined in terms of:
  (i) **Precision**: It is accuracy of the fractional part of the value. It measured as number of bits.
  (ii) **Range**: It is combination of the range of fractions and exponents.

3.2.1.3 Complex

- Some languages such as Python and FORTRAN language support a complex data type.
- Each value consists of two parts: 1) The real part, and 2) The imaginary part.

- These parts are Floating. For example, in Python, Literal form: 12 + 2.
  Here, Real part is 12 and Imaginary part is 2.
- In some languages, Complex number is treated as collection of data types like LIST, SET or in tabular format with rows and columns, and so on.

3.2.1.4 Decimal

- Decimal integer stores a fixed number of decimal digits, with decimal point at a fixed position, in Binary Coded Decimal (BCD) form.
- It takes at least 4 bits to code a decimal digit. For example, to store 6 digit coded decimal, it takes 24 bits but in binary, it takes only 20 bits.
- This is primary data type used for business data processing applications (money). So, it is essential for COBOL language.
- A decimal data type is also offered in C#.
- The decimal operations are done in hardware on machines, otherwise simulated in software.

Advantages:

1. These are capable of storing decimal values within range which is not possible in floating point values.
2. Accuracy is provided for decimal values.

Disadvantage:

1. Limited Range as no exponents is allowed.

3.2.2 Boolean Types

- Boolean data types are the simplest of all data types.
- This type was introduced by ALGOL 60. This type is used to represent switched and flags in programs.
- One exception is C89, numeric expressions are used as conditionals. All operands with nonzero values are considered true and zero is considered false in such expressions.
- Range of values of Boolean data type are two values, one for "**true**" and one for "**false**".
- It could be implemented as single addressable unit such as byte, word, bits.
- It can be given as enumeration explicitly.
- Use a particular bit for a value. For example, last bit: 0 represents False, 1 represents True.
- Use the entire storage then 0-False otherwise 1-True.
- Basic operations are: NOT, AND, and OR.

Advantage:

1. In this type, readability is more.

3.2.3 Character Types

- Character types stored as numeric coding (ASCII/Unicode).
  - Most commonly used coding is 8 bit code ASCII (American Standard Code for Information Interchange).
  - **Unicode** (16-bit coding) is an alternative. Most natural languages characters are included.
- It contains single character as a value of a data object.
- **Collating sequence**: The ordering of characters used for lexicographic sorting.
  - Originally it is used in Java. C# and JavaScript also support Unicode.
  - Implementation is supported by the underlying hardware.

Operations:

- Relational operations and Assignment operations.
- Testing the type of character (Digit, Letter, Special symbol).

3.3 Character String Types

- Values are sequences of characters in character string.
- Constants are used to label output.
- Input and output of all kinds of data are done in strings.
- Character strings are an essential type for all programs performing character manipulation.

3.3.1 Design Issues

- Two most important design issues are as follows:
  - Is it a primitive type (No array style subscripting operation) or just a special kind of character array?
    For example, `char * s`? OR `String s`?
  - Is the length of strings static or dynamic?
    For example,

```
char *s[];
string s[10];
```

3.3.2 Strings and their Operations

- If strings are not defined as a primitive type then data is stored in arrays of single characters.
- For example, Pascal, C, C++, Ada languages has this approach.

Typical operations:

- **Assignment** (=) as well as **Copying string** (For example, strcpy).
- **Comparison or Relational operations**: Equal (+), Less than(<), Greater than(>) (or with combination of these or strcmp).
- **Concatenation:** Appends two strings (For example, strcat or + in Java).

- **Substring reference**: Uses positioning subscript.
- **Pattern matching:** For substring selection (strstr).
- **Input/Output formatting**.

Dynamic strings:

- String is evaluated at runtime.
- In Ada, string is predefined to be single dimension array of character elements.
- Substring reference allows substring of a string treated as a value in a reference or a variable in an assignment. It is denoted by parenthesized integer range indicating desired substring by character position. For Example, Name (3:5); this specifies substring of third, fourth and fifth characters of the value in Name.
- String concatenation is specified by ampersand (&). For Example, Name=Name &surname; Here, surname is appended to name (To the right end of the name).

Character String Type in Certain Languages:

- In **C and C++** language:
  - Not primitive type.
  - Use char arrays to store character strings and a library of functions whose header is `string.h` which provide operations.
    For example, `char *str="Orange";`
  - Character strings are terminated with a special character, null represented with zero.
  - `str` is a character pointer set to point at character string `Orange0`. Here, zero(`0`) is NULL character.

Library Functions:

- **Strcpy():** Copy strings.
- **Strcat():** Concatenates one given string with another.
- **Strcmp():** Compares two strings.
- **Strlen():** Returns number of characters in string excluding NULL character.

- **In SNOBOL4:**
  - String is the primitive type.
  - SNOBOL4 is a one of the string manipulation language.
  - Many operations, including elaborate pattern matching built into the language.
- **In Fortran77, Fortran90, Basic and Python:**
  - String is Primitive type.
  - It provides assignment, relational operators, concatenation and substring reference operations.

- **In Java:**
  - The String class whose values are constant strings.
  - The String Buffer class whose values are changeable called mutable.
  - String Buffer variables allows subscripting.
- **In Perl, JavaScript, Ruby, and PHP:**
  - It provides built-in pattern matching using regular expressions.

3.3.3 Character String Length Options

- There are several design choices for the length of string values.

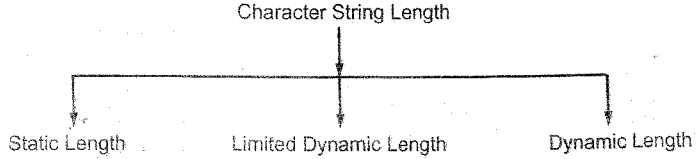


Fig. 3.3: Design choices for the length of string values

- **Static Length Strings:**
  - Length is static, specified in the declaration part.
  - Examples: COBOL, PASCAL, FORTRAN90, ADA, Java's String class.
  - Static length strings are full.
  - If shorter string is assigned then empty characters are set to blanks.
- **Limited Dynamic Length Strings:**
  - It allows strings of varying length up to declared and fixed maximum set by the variable's definition.
  - It can store any number of characters between Zero and Maximum. Examples: C and C++.
  - A special character is used to indicate the end of a String's characters rather than maintaining the length.
- **Dynamic Length Strings:**
  - It allows strings of varying length (no maximum).
  - **Example:** SNOBOL4, Perl, JavaScript.

Advantage:

1. It provides maximum flexibility.

Disadvantage:

1. Overhead of dynamic storage allocation and de-allocation.
2. Ada supports all three string length options.

3.3.4 Evaluation

- Help to writability of a language.
- As a primitive type with static length, they are inexpensive in terms of language and compiler complexity.

- The standard libraries of string manipulation subprograms remove deficiency when strings are non-primitive types.
- Most flexible is dynamic length. The overhead of implementation must be weighed against flexibility.

3.3.5 Implementation of Character String Types

- These types sometimes support hardware.
- Mostly software is used to implement string storage, retrieval and manipulation.
- As we discussed above, there are three design choices for the length of string values.

Static Length:

- Compile-Time Descriptor for Static string has following three fields:
  1. **Static string:** Every descriptors first field is the name of the type.
  2. **Length:** The Second field is the type's length.
  3. **Address:** The Third field is the address of the first character.

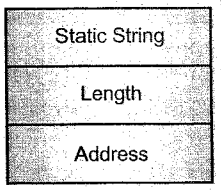


Fig. 3.4: Compile-Time Descriptor for Static string.

Limited Dynamic Length:

- It requires a run-time descriptor to store fixed maximum length and current length.
- C and C++ do not require run time descriptors because the end of the string is shown by the NULL character.
- Do not need maximum length as index values in array references are not checked range.
- When string variable is bound to storage, sufficient storage for maximum length is allocated.
- The maximum length is fixed at compile time.

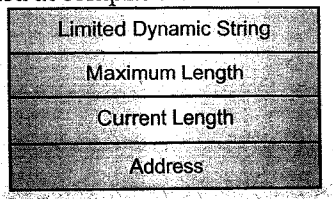


Fig. 3.5: Run-Time Descriptor for Limited dynamic String

Dynamic length:

- Dynamic Length requires more complex storage structure.
- The length of a string and the storage must grow and shrink dynamically.
- It needs run-time descriptor as only current length needs to be stored.

Approaches to Dynamic allocation and Deallocation problem:

- There are two possible approaches to Dynamic allocation and Deallocation problem.

1. **Strings can be stored in a linked list.**
   - When a string grows, newly required cells can come from anywhere in the Heap.

Advantages:

1. Allocation and De-allocation processes are simple.

Disadvantages:

1. It requires more storage due to links in the list representation.
2. Due to pointer chasing, some string operations are slowed.

2. **Complete strings can be stored in adjacent storage cells.**
   - The problem occurs when a string grows.
   - Storage adjacent to the existing cells continues to be allocated which is not always available.
   - In this case, new area of memory is found which will store new string. Old part is moved to this new area.
   - The memory cells used for old string are deallocated.

Advantages:

1. Operations on strings are much faster.
2. Less storage is required to store strings.

Disadvantages:

1. Slower Allocation.
2. Problem of managing allocation and deallocation of variable size segments.

3.4 User-defined Ordinal Types

- The range of possible values can be easily associated with the set of positive integers in ordinal type.
- Examples of primitive ordinal types in Pascal, Ada and Java are integer, char and boolean.

3.4.1 Enumeration Types

- All possible values, which are named constants are provided or enumerated in the definition.
- It provides a way of defining and grouping collections of named constants called as **enumeration constants.**

Example of enumeration type:

- **In C#:**

```
enum days {mon, tue, wed, thu, fri, sat, sun};
```

- **In Ada:**

```
type DAYS is (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday,
Sunday);
```

3.4.1.1 Design Issues

- The designing issues for enumeration types are as follows:
  - Whether an enumeration constant allowed appearing in more than one type definition? And if so, how is the checking of the type of an occurrence of that constant?
  - Whether enumeration values coerced to integer?
  - Ada allows using more than one declaration with the help of the concept "Overloaded Literals". Is this allowed in other language?
  - Whether there is any other type coerced to an enumeration type?

3.4.1.2 Design

- Enumeration type variables can be used as:
  - Array subscript.
  - Variables in *for* loop.
  - Expressions of *case* selector.
  - But not input or output.
- Two variables and/or literals of the same type compared with the relational operators, with relative positions in the declaration determining the result.

Example:

```
type ctype =( red, green, blue, yellow, black);
var colour: ctype;
 ...
colour:= black;
if colour > green ...
```

Boolean Expression in *if* will evaluate to true.

- **In ANSI C and C++:**
  - Literal constant cannot appear in more than one enumeration type definition in given referencing environment.
  - Values are implicitly converted to integer subject to the rules of use of integer.
  - For example, we might use `0` to represent blue, `1` to represent red, and so forth. These values could be defined as follows:

    ```
    int red = 0, green = 1;
    ```
  - **In C++**, we could have the following:

    ```
    enum colour {red, green, blue, yellow, black, white};
    colour myColour = green, yourColour = red;
    ```
- **In Ada:**
  - Enumerated type is similar to Pascal except that literals are allowed to appear in more than one declaration in the same referencing environment called as an **overloaded literal**.

  - To resolve overloading (Deciding the type of an occurrence of such a literal), the rule is it must be determinable from the context of its appearance.
  - BOOLEAN and CHARACTER are predefined Enumerated types.
  - Operations are for predecessor, successor, position in list of values, value for specific position number.
  - Example: `enum day {sat = 6, mon=2, wed=4, tue=3, fri=5, sun=1 };`
- **In Java :**
  - Enumeration types (Enums) are used to create own data type like classes. Enum implements an interface in Java.

3.4.1.3 Evaluation of Enumerated Type

- Enumerated data type provides many advantages in Readability and Reliability. Appropriately named enumerated types improve the readability of the code.
- Languages such as C++ and Ada, treat each enumerated type as distinct from integers and other enumerated types. It enables the compiler to catch a variety of possible errors.
- **Readability:**
  - Enhanced in a very direct way.
  - Named values are easily recognized, but coded values are not. For example, there is no need to code a color as a number.
- **Reliability:**
  - **Compiler can check:**
    - No arithmetic operations are legal on enumeration types. For example, string colors are not allowed to be added.
    - Enumeration variable can be assigned a value only within its defined range. For example, if the colors enumeration type has 5 enumeration constants and uses 0. . 4 as its internal values, number greater than 4 can not be assigned to a colors type variable.
    - Better support for enumeration is provided by Java 5.0, Ada and C# than C++ as enumeration type variables are not coerced into integer types in these languages.
  - Enumerated type is more readable.
  - It also provides Type checking.

3.4.2 Subrange Types

- An ordered, contiguous subsequence of an ordinal type is called a Subrange type.
  Example: Subrange of integer type is `12..18`.
- Subrange types are introduces by Pascal and included in Ada also.

3.4.2.1 Ada's Design

- Subranges are included in the class of types called Subtypes. Subtypes are not new types, but only new names for possibly restricted or constrained versions of existing types.

- For example, Ada's Design:

```
type Days is (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday,
Sunday);
subtype WkEnd is Days range Saturday.. Sunday
subtype WkDays is Days range Monday..Friday;
subtype Indices is Integer range 1..50;
```

In above examples, restriction is on range of possible values.

- Operations defined for parent type are also defined for the subtype, except assignment of values outside the specified range.
- For Example,

```
Day1: Days;
Day2: WkDays;
Day2:= Day1;
```

- The assignment is legal, unless `Day1` value is `Saturday` or `Sunday`.

3.4.2.2 Subrange Evaluation

- Subrange types help to readability by making it clear to the readers that variables of subtypes can store only certain range of values.
- Reliability is increased with subrange types because assigning a value to a subrange variable that is outside the specified range is detected as an error either by compiler (e.g. assigned value become literal value) or by the runtime system (case of variable or expression).

3.4.3 Implementation of User-defined Ordinal Data

- Enumeration types are implemented as non-negative integers.
  - First enumeration value is represented by 0, second by 1 and so on.
  - Operations allowed are different than integer except for relational operators.
  - In ANSI C and C++, enumeration types are treated as integer.
- To restrict assignments to subrange variables, subrange types are implemented like the parent types with code inserted by the compiler.
  - This increases code size and execution time but worth the cost.
  - A good optimizing compiler can optimize away some of the checking.

3.5 Array Types

- Array is composite type.
- So, first we learn about composite type then about array.

Composite Types:

- Non-scalar types are called as **Composite / Constructed types.**

- These are generally created by applying a type constructor to one or more simpler types.

Examples:

- **Record**: In COBOL, record is introduced. Consist of collection of data types. Same as structure.
- **Variant records/Union**: Different from normal record that only one field is valid at a time.
- **Arrays**: Collection of homogeneous elements stored in adjacent memory locations.
- **Sets**: Collection of distinct elements of base type.
- **Pointers**: Implemented as address i.e. l value.
- **Lists:** Sequence of elements but without indexing.
- **Files:** Represents data on storage devices.

3.5.1 Array Basics

- An array is an aggregate of homogeneous data elements in which an individual element is identified by its position in the aggregate, relative to the first element.
- The individual data elements of an array are of the same type. References to individual array elements are specified using one or more non constant subscripts.
- If the subscript expressions include variables, then requires an addition run-time calculation to determine the address of the memory location being referenced.
- Each data element of an array is of previously defined types either primitive or otherwise.

3.5.2 Array Design Issues

- The primary design issues specific to arrays are as follows:
  - What types are legal for subscripts?
  - Whether subscripting expressions in element references range checked?
  - When are subscript ranges bound?
  - When does array allocation take place?
  - What are the maximum numbers of subscripts allowed?
  - Can arrays be initialized when they have their storage allocated?
  - If any, what kinds of slices are allowed?
  - Whether ragged or rectangular multidimensional arrays allowed, or both?

3.5.3 Array and Indices

- Indexing (or subscripting) is a mapping from indices to elements.
- Dynamic selector of an array, which consist of one or more items known as Subscript or Indexes.
- The mapping can be shown as:

  `array_name (index_value_list)` → an element
- In Index Syntax, FORTRAN, PL/I, Ada use parentheses.

- Ada uses parentheses explicitly which shows uniformity between array references and function calls as both are mappings. Most of other languages use braces (brackets).
- **Example of ADA assignment statement:**

```
Total:=Total + B(I);
```

- As ( ) are used for subprogram parameters and array subscripts in Ada, which results in reduced readability.
- C - based languages use [ ] to delimit array indices.
- Two distinct types are involved in an array type:
  - The element type.
  - The type of Subscript. This is mostly a sub-range of integers.
- In Ada, as the subscripts other types are allowed. For example: Boolean, char, and enumeration.
- Java, ML, and C# do specify range checking of subscripts but C, C++, Perl, and FORTRAN do not specify range checking of subscripts.
- In Perl, all arrays begin with at sign (@) as array elements are scalars and the names of scalars always begins with dollar sign($), references to array elements use dollar($) signs rather that at(@) signs in their names. For example, for the @list, the second element is referenced with $ list[1].

3.5.3.1 Arrays Index (Subscript) Types

- **C, FORTRAN:** Array indices are Integer types only.
- **Ada:** Array indices are integer types or enumeration types includes Boolean and char.
- **JAVA:** Contains array indices as Integer types only.
- Index range checking for different languages is as follows:
  - Range checking is not specified in C, C++, Perl, and FORTRAN.
  - Range checking is specified in Java, ML, C#.
  - By default requires range checking, but it can be turned off in ADA.

3.5.4 Subscript Binding and Array Categories

- The binding of subscript type to an array variable is Static, but the subscript value ranges are sometimes dynamically bound.
- The lower bound of all index ranges is 0 in C-based languages.

Categories of Array Memory Allocation:

- There are five categories of Array Memory Allocation, based on binding to:
  - Subscript Ranges.
  - Storage.
  - ROM.

1. **Static array:**
   - Subscript ranges are statically bound and storage allocation is static (done before run-time).

Advantage:
- Efficiency (No dynamic allocation and deallocation).

Example: Arrays which include static modifier are static in C and C++.

2. **Fixed stack-dynamic array:**
   - Statically bound to subscript ranges.
   - The storage allocation is done at declaration elaboration time during execution.

Advantage:
- Space is utilized efficiently. One subprogram's large array uses the same space as different subprograms large array can use.

Example: In C & C++ function, arrays declared without the static modifier are fixed stack-dynamic arrays.

3. **Stack-dynamic array:**
   - Dynamically bound to subscript ranges.
   - The storage allocation is dynamic means done at runtime i.e. "during execution".
   - They remain fixed during the lifetime of the variable once bound.

Advantage:
- Flexible as the size of an array need not be known in advance until the array is to be used.

4. **Fixed heap-dynamic array:**
   - It is like fixed stack-dynamic array.
   - Subscript ranges are dynamically bound, and the storage allocation is dynamic, but fixed after storage is allocated. When requested binding is done. Heap is used for Storage allocation, not stack.
   - When the user program requests them, the bindings are done during execution, instead at elaboration time.

Example:
- Fixed heap-dynamic arrays are provided in C & C++. malloc() and free() functions are used in C. In C++, new() and delete() operations are used.
- In C#, an Array class `ArrayList` provides fixed heap-dynamic.
- All non-generic arrays are fixed heap-dynamic arrays in Java. When created, keeps the same subscript ranges and storage.

5. **Heap-dynamic array:**
   - Binding of subscript ranges and storage allocation is done dynamically and it can change any number of times during the array's lifetime.

Advantage:
- **Flexibility:** Arrays grows or shrinks during program execution as the need for space changes by adding or deleting the elements in array.

Example:
- Generic heap-dynamic arrays in C# are objects of the C# List class. These array objects are created without any elements, as follows:

```
List <String> stringList = new List <String>();
```

- Add method is used to add elements to this object as follows:

```
stringList.Add("Shree");
```

- Java includes a generic class similar to C#'s List, named `ArrayList`.
- A JavaScript and Perl array supports heap-dynamic array to grow with the Push, which puts one or more new elements on the end of the array and unshift, which puts one or more new elements on the beginning of the array.
- For example, creating an array of first five even numbers in Perl with

```
@evenlist = (2, 4, 6, 8, 10);
```

Later, with push function the array could be extended as follows:

```
push(@evenlist, 12, 14);
```

- Now the array's value is (2, 4, 6, 8, 10, 12, 14).
- Heap-dynamic arrays are also supported by Python, and Ruby.

3.5.5 Heterogeneous Arrays

- A heterogeneous array is the array in which the elements might be of the different type. In other words, an elements type need not be same.
- These arrays supported by JavaScript, Perl, Python and Ruby.

3.5.6 Array Initialization

- Some languages, at the time of storage allocation allow array initialization.
- Just a list of values that are put in the array in the order in which the array elements are stored in memory.
- FORTRAN uses the **DATA** statement, or put the values in /... / on the declaration.

```
Integer list (3)
data list /0, 5, 5/        // list is initialized to the values
```

- C, C++, Java, C# allow initialization of their arrays. For example,

```
int list [] = {10,20,30,40};
```

- By the compiler, length of the array is set.
- Character strings in C and C++ are implemented as arrays of characters. For example, `char name [] = "Swara";`
- Arrays of strings in C and C++ can also be initialized with string literals. For example, `char *names [] = {"Ankit", "Swara", "Sayali"};`
- Java initialization of String objects is as follows,

```
String[] names = {"Ankit", "Swara","Sayali"}
```

- Positions for the values in Ada can be specified as follows,

```
MARKS: array (1..20, 1..2):= (1 => (24, 10), 2 => (10, 7), 3 =>(12, 30),
others => (0, 0));
```

- Pascal does not allow array initialization.

3.5.7 Array Operations

- The most common array operations are as follows:
  - Assignment.
  - Concatenation.
  - Slices.
  - Equality and inequality Comparison.
  - Transpose, Union of 2-dimensional array.
- The C-based languages do not provide any array operations, except methods of Java, C++, and C#.
- Array assignment as well as concatenation is allowed in Ada.
- Perl supports array assignments but does not support comparisons.
- Python's arrays are called Lists, even though they have all the characteristics of dynamic arrays. These arrays are Heterogeneous as the objects can be of any types.
- Python supports array assignments (only for reference changes), array concatenation and element membership operations.
- Ruby provides array concatenation.
- FORTRAN provides elemental operations as they are between pairs of array elements. For example, Using + operator between two arrays, we get an array of the sums of the element pairs of the two arrays.
- APL provides the most powerful array processing operations for vectors and matrices as well as unary operators (for example, to reverse column elements).
- There are some operations which we can perform with array are: Transpose of matrix (2-d array), Reverse array, Sort array, Add or any arithmetic operation with array, Slice or Subarray.

3.5.8 Rectangular and Jagged Arrays

- According to shape, array is categorized into Rectangular & Jagged array.
- A **Rectangular array** is a multi-dimensioned array in which all of the rows have the same number of elements and all columns have the same number of elements.
- A **Jagged array** is one in which all of the rows need not be the same number of elements.
- A jagged matrix has rows with different number of elements that means each row might have different number of elements in a matrix.
- For example, a jagged matrix may consist of three rows, one with 2 elements, one with 5 elements, and one with 9 elements.
- It is possible when multi-dimensioned arrays appear as arrays of arrays.
- C, C++, and Java support jagged arrays but not rectangular arrays.
- In some languages, a reference to an element of a multi-dimensional array uses a separate pair of brackets for each dimension. For example, `myArray[3][7];`
- C# and F# support rectangular arrays and jagged arrays.
- For rectangular arrays, all subscript expressions in references to elements are placed in a single pair of brackets. For example, `myArray[3,7]`

- Fortran and Ada support rectangular arrays.
- Following figure shows 2 different types of arrays:

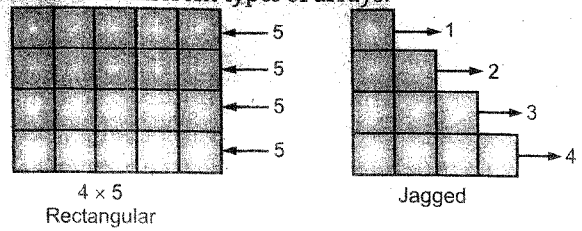


Fig. 3.6: **Rectangular and Jagged Arrays**

3.5.9 Slices

- A slice of an array is some substructure of an array.
- It is a referencing mechanism of an array as a unit.
- If arrays cannot be manipulated as units in a language, that has no use for slices.
- Slices are useful in those languages which have array operations.

Examples:

- In **Python**,

```
vector = [2, 4, 6, 8, 10, 12, 14, 16]
mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

  - `vector (3:6)` is a three-element array, which is [8, 10, 12].
  - `mat[0][0:2]` is the first and second element of the first row of mat is [1, 2].
- **Ruby** supports slices with the `slice()` method.

```
list = [2, 4, 6, 8, 10]
```

  - `list.slice(2,2)` returns the third and fourth elements of list: [6, 8]
  - `list.slice(1..3)` returns [4, 6, 8]
- Slice Examples in **Fortran 95**:

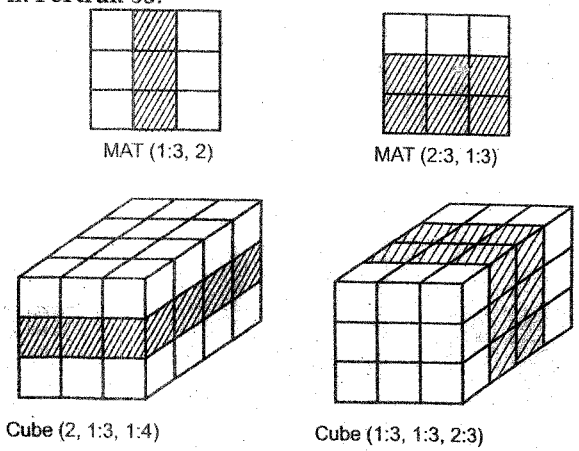


Fig. 3.7: **Slice examples in Fortran 95**

3.5.10 Evaluation

- Arrays have been included in virtually all programming languages. It is well developed as well as simple.
- The primary advances since their introduction in Fortran has been slices and dynamic arrays.
- It is convenient to allow the programmer to specify a particular substructure of multi-dimensional array; the increased difficulties of implementation and readability are not necessarily worth the convenience.
- The latest advances in arrays appeared in associate arrays.

3.5.11 Implementation of Arrays

- Access function maps subscript expressions to an address in the array.
- A single-dimensioned array is implemented as a list of adjacent memory cells.
- The address to be accessed by a reference such as, `list[k]`.
- Suppose the array list is defined to have a subscript range lower bound of `0`.
- The access function for list is often of the form :

```
address(list[k]) = address (list[0])+ k* element_size
```

- Access function for single-dimensioned arrays:

```
address(list[k])= address(list[lower_bound])+((k-lower_bound)*
                                                element_size)
```

- The compile-time descriptor for single-dimensioned arrays can have the form shown in the Fig. 3.8.

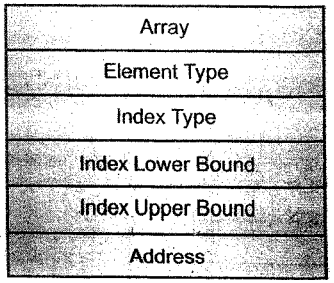


Fig. 3.8: **Compile-time Descriptor for Single-dimensioned Arrays**

- The compile-time descriptor for Multi-dimensioned arrays can have the form show in the following figure.

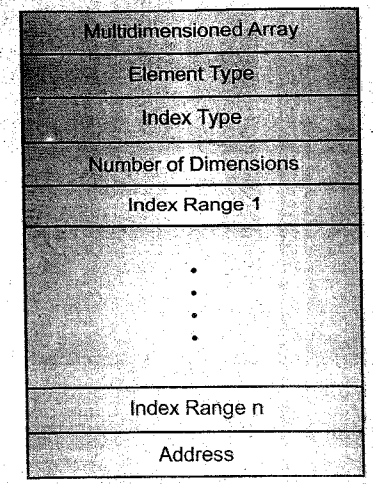


Fig. 3.9: Compile - time Descriptor for Multi-dimensioned Arrays

Accessing Multi-dimensioned Arrays:

- Following two common ways are to access Multi-dimensional arrays:
  - Row major order (by rows) which is used in most languages.
  - Column major order (by columns) which is used in Fortran.

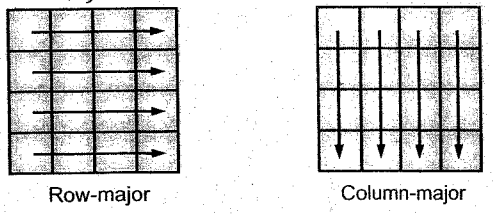


Fig. 3.10: Row major order and Column major order

- For example, if the matrix had the values:

  3 4 5
  6 7 8
  1 2 9

  - It would be stored in row major order as: 3, 4, 5, 6, 7, 8, 1, 2, 9.
  - If stored in column major, order in memory will be: 3, 6, 1, 4, 7, 2, 5, 8,9.
- In all cases, sequential access to matrix elements will be faster if accessed in the order in which they are stored, as that will minimize the paging. (Paging means moving blocks of information between disk and main memory. The objective is to keep the frequently needed parts of the program in memory and the rest on disk.).
- Locating an element in a Multi-dimensioned Array (row major):

```
location (a[i, j]) = address of a [0,0] + ((i* n)+ j)) * element_size.
```


3.5.12 Additional Related Points of Arrays

1. **Row Pointers Layout:**

- We can allocate pointers to access arrays called Row pointers to avoid allocating holes in memory.

Example:

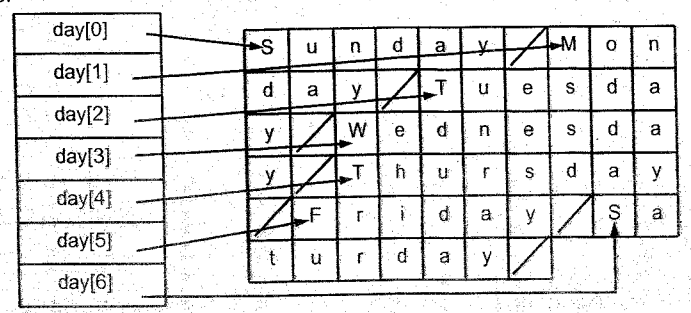


Fig. 3.11: Row Pointers Layout

- Required allocation size for,
  - Array = 57 bytes
  - Pointers = 28 bytes
  - Total Space = 85 bytes

2. **Contiguous Memory Allocation:**

- Alternative way to allocate array is **Contiguous allocation**. Here, each element in array has a row of allocated space.

S	U	N	D	A	Y				
M	O	N	D	A	Y				
T	U	E	S	D	A	Y			
W	E	D	N	E	S	D	A	Y	
T	H	U	R	S	D	A	Y		
F	R	I	D	A	Y				
S	A	T	U	R	D	A	Y		

Memory holes

- So, Required space to store Array = 70 bytes

Fig. 3.12: Contiguous Memory Allocation

3. Here, are some differential points between both layouts:

Table 3.1: Difference between Contiguous Layout and Row Pointer Layout

Sr. No.	Contiguous layout	Row pointer layout
1.	Each element is present at separate row	Pointers used for each element to access.
2.	Memory holes present.	No memory holes.
3.	Requires less space than row pointer layout.	More space required.
4.	Slow accessing.	Fast accessing.
5.	Example: Simple row major, column major	Example: Row pointed major, column pointed major

4. **Dope Vector:**

- Dope Vector is called as Run Time Descriptor.
- When the size and bounds are not known at compile time, then the compiler must arrange them to be available when the compiled program needs to compute an address at run time. The mechanism is called as **dope vector**.
- The dope vector for an array of dynamic shape is generally placed next to the pointer to the array in the fixed part of the stack frame.

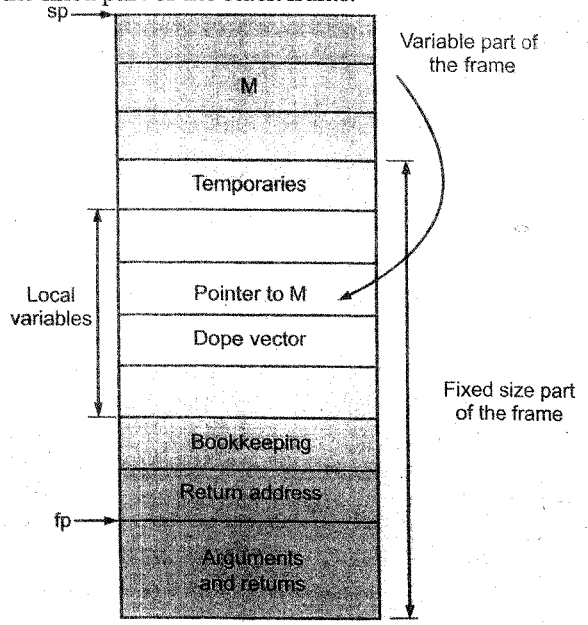


Fig. 3.13: Dope Vector

Conformant Array:

- A Conformant Array is not a dynamic array that resizes itself to fit data.

Example:

```
main()
{
int x[4]={1,2,3,4};
F(x);
}
void f( int y[])    // y is conformant array.
{.....
}
```

3.6 Associative Arrays

- An associative array is collection of an unordered data element that is indexed by an equal number of values called **keys**.
- User-defined keys must be stored. So each element of an associative array is a pair of entities, a Key and a value.
- Associative arrays are supported by the standard class libraries of Java, C++, C#, and F#.
- In Python, the dictionaries are similar to Perl except the values are all reference to objects.
- In PHP, arrays are both normal arrays and associative array.
- Through a .NET class, C# and F# support associative arrays.
- In PHP, associative array looks like,

```
$names=array(1=>"Shardul",2=>"Sharayu",3=>"Avdhoot",4=>"Smaran",
5=>"Riddhi",6=>"Siddhi");
```

3.6.1 Design Issues

Design issues of associated arrays are:

- Is the size of an associative array static or dynamic?
- What is the form of references to elements in an associative array?

3.6.2 Structure and Operations in Perl

- With % names starts. Literals are delimited with the parenthesis.
  For example,

```
%hi_temps = ("Samruddhi" => 12, "Sarthak" => 8, "Shree"=>2,...);
%salaries= ("Seeta" => 75000, "Geeta" => 57000,"Meeta" =>55750, "Neeta" =>
47850);
```

- Alternative notation is:

```
%hi_temps = ("Monday", 77, "Tuesday", 79,"Wednesday", 83);
%salaries= ("Seeta", 75000, "Geeta", 57000,"Meeta",55750, "Neeta", 47850);
```

- Using braces and keys subscripting is done.
  - **Example:** `$hi_temps{"Wed"} = 83;`
    #returns the value 83
  - **Example:** An assignment of 58850 to the element of % salaries with the key "Reeta" would appear as follows:
    `$salaries{"Reeta"} = 58850;`
- Elements can be removed with delete.
  - **Example:** `delete $hi_temps{"Tue"};`
  - **Example:** `delete $ salaries{"Seeta"};`
- A new elements is added by,
  `$hi_temps{"Thurs"} = 91;`

3.6.3 Implementing Associative Arrays

- Implementation of Perl's associative array is optimized for fast lookups. It also provides relatively fast reorganization when array growth required.
- A 32-bit hash value is computed for each entry and stored with entry, even though an associative array initially uses a small part of the hash value.
- When an associative array must be expanded beyond its initial size, hash function remains unchanged; instead more bits of hash values are used.
- In the above case, half of the entries must be moved.
- Expansion of an associative array is not free but also not as costly as someone thinks.

3.7 RECORD TYPES

- A heterogeneous aggregate of data elements is called **record.**
- By names the individual elements are identified.
- In C, C++, and C#, records are supported with the structure data type.
- Structures are a minor variation on classes in C++.

Design issues:

- What is the syntactic form of references to the field in record types?
- Whether elliptical references allowed?

3.7.1 Definitions of Record

- The basic difference between a record and an array is that elements or fields in records are not referenced by indices. Instead, the fields are named with identifier, and references to the fields are made using these identifiers.

3.7.1.1 Definitions of Record in COBOL

- COBOL uses level numbers to show nested records; others use recursive definition.

- A record declaration, which is part of the data division of a COBOL program, is illustrated in the following example:

```
01 EMPLOYEE-RECORD.
02 EMPLOYEE-NAME.
04 FIRST PICTURE IS X(25).
04 MIDDLE PICTURE IS X(15).
04 LAST PICTURE IS X(25).
02 HOURLY-RATE PICTURE IS 99V99.
```

- The numbers 01, 02 and 04 at the lines beginning of the record declaration are level numbers. This indicates their relative values in the hierarchical structure of the record.
  - `PICTURE` clause: Specify the formats of the field storage location.
  - `X(25)`: Specify 25 alphanumeric characters.
  - `99V99`: Specify four decimal digits with decimal point in the middle.

3.7.1.2 Definitions of Record in Ada

- Record structures are indicated in an orthogonal way.

```
type Employee_Rec_Type is record
First: String (1..20);
Mid: String (1..10);
Last: String (1..20);
Hourly_Rate: Float;
end record;
Employee_Rec:Employee_Rec_Type;
```

3.7.2 References to Records

- References to the individual fields of records are syntactically specified by several different methods, two of which are,
  1. Name the desired field.
  2. Enclosing records.
- **Record field references in COBOL:**
  `field_name OF record_name_1 OF... OF record_name_n`
  For example, the Middle field in the COBOL record example above can be reference with, `Middle OF EMPLOYEE-NAME OF EMPLOYEE-RECORD`
- **Record field references in Others (dot notation):**
  `record_name_1.record_name_2.... record_name_n.field_name`
  - Most language use dot notation. For example,
    `Employee_Record.Employee_Name.Middle`

- Fully qualified references must include all record names.
- As long as the reference is unambiguous, elliptical references allow leaving out record names.
- For example in COBOL FIRST, FIRST OF EMPLOYEE-NAME and FIRST of EMPLOYEE-RECORD are elliptical references to the employee's first name.

3.7.3 Operations on Records

- Assignment is very basic operation. Pascal, Ada, and C++ allows assignment if the types are identical.
- Record comparison is allowed in ADA. For comparison = and /= is used. One operand can be an aggregate in record comparison.
- Using an aggregate, initialization is allowed in Ada.
- COBOL supports MOVE CORRESPONDING. It copies a field of the source record to the corresponding field in the target record.

3.7.4 Evaluation

- Records and arrays are closely related structural forms.
- Arrays are used when all the data values have the same type and/or are processed in the same way.
- Records are used when the collection data values are heterogeneous and the different fields are not processed in the same way. Also, the fields of a record need not be processed in a particular order.
- As subscripts are dynamic, access to record fields is much faster than access to array elements (field names are static).
- Dynamic subscripts could be used with record field access, but it would disallow type checking and would be much slower.

3.7.5 Implementation of Record Types

- The fields of records are stored in adjacent memory locations.
- A record storage representation consists of a single sequential block of memory in which the components are stored in sequence.
- Individual components may need descriptors to indicate their data type and other attributes.
- Using Offsets field accesses are all handled. The Offset address is associated with each field if relative to the beginning of the records.
- No runtime descriptor for the record is required.
- The compile-time descriptors for a record have the general form shown in the following Fig. 3.14.

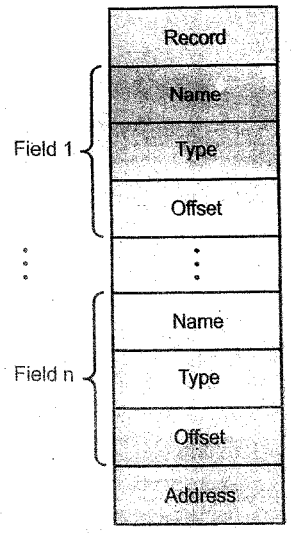


Fig. 3.14: A Compile-time Descriptor for a Record

3.8 Unions Types

- A union is a type whose variables are allowed to store different type values at different time during execution.

3.8.1 Design issues

- The major design issues of union types are:
  - Should type checking be required?
  - Should unions be embedded in records?

3.8.2 Discriminated vs. Free Unions

- Union constructs in which there is no language support for type checking is provided by Fortran, C, and C++. Such kind of union in these languages is called **free union**.

Example:

```
union flexType
{
   int intE1;
   float floatE1;
};
union flexType el1;
   float p;
      ...
el1.intE1 = 30;
p= el1.floatE1;      // assign 30 to float variable p
```

- This last assignment is not type checked, as the system cannot determine the current type of the current value of el1, so it assigns the bit string representation of 30 to float variable p.

A Discriminated Union:

- Unions in which type checking require that each union include a type indicator known as a Discriminated Union.
- Ada supports discriminated union. Consider the code given below,

```
type Colors is (Red, Green, Blue);
type Shape is (Circle, Triangle, Rectangle);
type Figure (Form: Shape) is record
 Filled: Boolean;
   Color: Colors;
case Form is
when Circle => Diameter: Float;
when Triangle =>Leftside, Rightside: Integer;
              Angle: Float;
when Rectangle => Side1, Side2: Integer;
end case;
end record;
```

- ALGOL 68 was the first language which provides discriminated union.
- These are supported by ML, Haskell, and F#.

3.8.3 Evaluation

- Free unions are unsafe.
- They do not allow type checking.
- This is potentially unsafe construct.
- There is one of the reasons why C and C++ are not strongly typed.
- Java and C# do not support unions.
- In programming language, reflective of growing concerns for safety.
- Ada's, Algol 68's discriminated unions are safe.

3.8.4 Implementation of Union Types

- Unions are implemented by simply using the same address for every possible variant.
- During translation, the amount of storage required for the components of each variant is determined.
- Storage is allocated in the record for the largest possible variant.
- Each variant describes a different layout for the block in terms of number and types of components.
- During execution, no special descriptor is needed for a variant record as the tag component is considered just another component of the record.

- Consider Ada Example,

```
type Node (Tag : Boolean) is
   record
   case Tag is
      when True => Count : Integer;
      when False => Sum : Float;
   end case;
   end record;
```

- The descriptor for this type could have the form shown in Fig. 3.15.

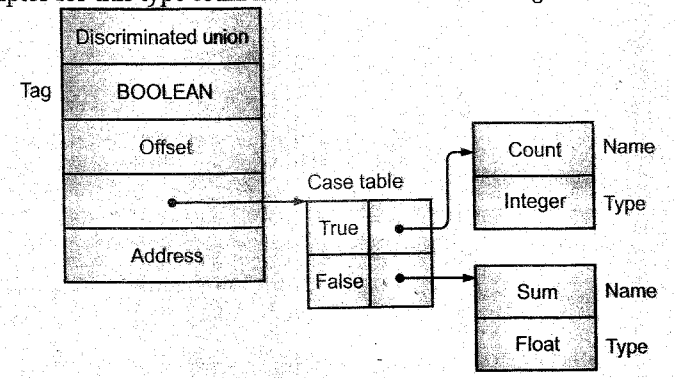


Fig. 3.15: Compile Time Descriptor

3.9 POINTER AND REFERENCE TYPES

- A pointer type in which the variables have a range of values that consists of memory addresses and a special value, *nil*.
- The value *nil* is invalid address, used to indicate that a pointer cannot be used currently to reference any memory cell.
- Pointers are designed for two distinct kinds of uses:
  - Provide the power of indirect addressing.
  - Provide a way to manage dynamic memory.
- A pointer can be used to access a location in the area where storage is dynamically created (usually called a heap).

3.9.1 Design Issues

- The primary design issues particular to pointers are the following:
  - What are the scope of and lifetime of a pointer variable?
  - What is the lifetime of a heap-dynamic variable?
  - Whether pointers restricted as to the type of value to which they can point?
  - Whether pointers used for dynamic storage management, indirect addressing, or both?
  - Should the language support pointer types, reference types, or both?

3.9.2 Pointer Operations

- A pointer type usually includes two fundamental pointer operations:
  1. **Assignment**: A pointer variable's value is assigned (or set) to some useful address.
  2. **Dereferencing**: A reference through one level of indirection is taken.
     - Dereferencing yields the value stored at the location which is represented by the pointer's value.
     - Dereferencing can be of explicit or implicit type.
     - In C++, dereferencing is explicitly specified with the (*) as a prefix unary operation.
- If `ptr` is a pointer variable with the value 9999, and the cell whose address is 9999 has the value 506, then the assignment.

```
j = *ptr;          //sets j to 506.
```

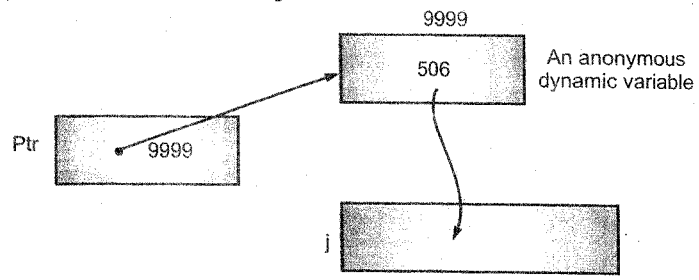


Fig. 3.16: The assignment operation j = *ptr

- In C and C++, there are two ways a pointer to a record can be used to reference a field in that record.
  - If a pointer variable p points to a record with a field name age. To refer to that field (*p).age can be used.
  - When the operator -> use between a pointer to a structure and a field of that structure combines dereferencing and field reference.
  - For example, the expression `p-> age` is equivalent to `(*p).age`
- Languages that provide pointers for the management of a heap must include an explicit allocation operation.
  - Allocation is sometimes specified with a subprogram, such as `malloc` in C.
  - In a language that supports object-oriented programming, allocation of heap objects is often specified with new operation.
- C++ does not provide implicit deallocation, so used delete as its deallocation operator.

3.9.3 Pointer Problems

- Following are the problems which occurs in pointers:
  1. **Dangling pointers (dangerous):** A pointer points to a heap-dynamic variable that has been deallocated.
  2. **Lost heap-dynamic variable:** An allocated heap-dynamic variable that is no longer accessible to the user program (often called garbage).

3.9.3.1 Dangling Pointers or Dangling Reference

- A pointer points to a heap-dynamic variable that has been deallocated.
- Dangling pointers are dangerous, the reasons are:
  1. The location being pointed to may have been allocated to some new heap-variable. If the new variable is not the same type as the old one, type checks of uses of the dangling pointer are invalid.
  2. Even though the new one is of same type, its new value will bear no relationship to the old pointer's dereferenced value.
  3. If the dangling pointer is used to change the heap-dynamic variable, the value of the heap-dynamic variable will be destroyed.
  4. Possible that the location now is being temporarily used by the storage management system, as a pointer in a chain of available blocks of storage, allowing a change to the location to cause the storage manager to fail.
- The sequence of operations that creates a dangling pointer in many languages is as follows:
  1. A new heap-dynamic variable is created and pointer p1 is set to point at it.
  2. p1's value is assigned to pointer p2.
  3. The heap-dynamic variable pointed to by p1 is explicitly deallocated (possibly setting p1 to nil), but p2 is not changed by the operation. Now p2 is a dangling pointer. If the deallocation operation did not change p1, both p1 and p2 would dangle. (Of course, this is a problem of aliasing as, p1 and p2 are aliases).
- For example in C++:

```
int* arrayp1;
int* arrayp2 = new int[50];
arrayp1 = arrayp2;
delete[] arrayp2;        // Now, arrayp1 is dangling.
```

- In C++, both arrayp1 and arrayp2 are now dangling pointers, as the C++ delete operator has no effect on the value of its operand pointer.
- Mostly wrong use of delete causes Dangling pointer.
- In the above code, if you try,

```
arrayp1[0]=10;          //it generates segmentation fault
```

- After delete [ ] arrayp2, it is good practice for making pointer as 0 using,

```
arrayp2 = NULL;
```

3.9.3.2 Lost Heap-Dynamic Variables

- A heap-dynamic variable that is no longer referenced by any program pointer "no longer accessible by the user program." Such variables are called garbage as not satisfying their main basic purpose and not reallocated for any new use in the program.
- Lost heap-dynamic variables are often created by the following sequence of operations:
  1. Pointer p1 is set to point to a newly created heap-dynamic variable.
  2. To point to another newly created heap-dynamic variable, pointer p1 is later set.
- Consider the code in C,

```
int a=2,b=3;
int *P;
P = &a;
.....
P = &b;
```

The first heap-dynamic variable is lost i.e. variable a inaccessible now.

- **Memory leakage**: Process of losing heap-dynamic variables is known as memory leakage.

3.9.4 Pointers in C and C++

- Use of pointers is highly flexible but it must be used with care.
- Pointers can point at any variable without considering when or where it was allocated.
- Pointers used for addressing purpose and dynamic storage management.
- Pointer arithmetic is possible in C and C++ makes their pointers interesting than the other programming languages pointers.
- In C and C++, pointers can point at any variable without considering the allocation. They can point at any variable anywhere in memory. And it is dangerous to use of such pointers.
- Dereferencing is explicit.
- In **C and C++**,
  - The asterisk (*) represents the dereferencing operation.
  - The ampersand (&) denotes the operator for producing the address of a variable.
- For example, in the following code,

```
int * ptr;
int count, i;
      ...
ptr = &i;        //variable ptr sets to the address of i
count = *ptr;    //Dereference ptr to produce the value at i,
                 //then assign to count
```


- The two assignment statements are equivalent to the single assignment.

```
count = i;
```

Example 1:

- Pointer Arithmetic in C and C++:

```
int list[10];
int* ptr;
```

Now consider the assignment,

```
ptr = list;
```

The address of `list[0]` copied to `ptr`.

- By means of above assignment, the following statements are true:
  - `*(ptr + 1)` is equivalent to `list[1]`
  - `*(ptr + index)` is equivalent to `list[index]`
  - `ptr[index]` is equivalent to `list[index]`
  - `*(ptr+5)` is equivalent to `ptr[5]` and `list[5]`

Example 2:

```
float stuff[100];
float *p;
p = stuff;
```

`*(p+10)` is equivalent to `p[10]` and `stuff[10]`.

`*(p+i)` is equivalent to `p[i]` and `stuff[i]`.

- C and C++ include pointers of type void *, point at values of any type. They are generic pointers in effect.
- Type checking is not a problem with void *pointers as these languages disallow dereferencing them.
- One common use of void *pointers is as the types of parameters of functions which operate on memory.
- Domain type not necessary be fixed (void *). void * points to any type. It can be type checked and cannot be de-referenced.

3.9.5 Reference Types

- A reference type variable is similar to a pointer, with important and fundamental difference: A pointer refers to an address in memory, while a reference refers to an object or a value in memory.
- A special kind of pointer type called a reference type is included in C++. In the function definition used primarily for formal parameters.
- In C++, reference type variable is a constant pointer always implicitly dereferenced.
- As C++, reference type variable is a constant. It must be initialized with the address of some variable in its definition. A reference type variable never set reference to any other variable once initialization is done.

- By preceding their names with ampersands (&) reference type variables are specified in definitions.

  For example,

```
int result = 0;
int &ref_result = result;
   ...
ref_result = 100;
```

- In this example, `result` and `ref_result` both are aliases.
- For increased safety over C++, the designers of Java removed C++ style pointer altogether.
- In Java, reference variables are extended from their C++ form that allows them to replace pointers entirely.
- The basic difference between Java references and C++ pointers is that Java reference variables refer to class instances and C++ pointers refer to memory addresses.
- As Java class instances are implicitly deallocated (there is no explicit deallocation operator), there is no dangling reference.
- C# includes the pointers of C++ and the references of Java. Strongly discouraged the pointers use. The unsafe modifier must be included if method uses pointers.
- All variables in the object-oriented languages such as Python, Ruby, Smalltalk, and Lua are references. They are always implicitly dereferenced. Also, the direct values of these variables cannot be accessed.

3.9.6 Evaluation

- Dangling objects garbage and dangling pointers are problems as is heap management.
- Pointers are like *goto* statements.
  - The *goto* statement widens the range of statements that can be executed next.
  - Pointer variables widen the range of memory cells that can be accessed by a variable.
- In some kinds of programming applications, pointers are essential.
  - For example, pointers are necessary for writing device drivers, in which specific absolute addresses must be accessed.
- The references of Java and C# provide the flexibility and the capabilities of pointers, without the hazards.
- It remains to be seen whether the programmers will be willing to trade the full power of C and C++ pointers for the greater safety of references.
- Pointers or references are necessary for dynamic data structures as we can't design a language without them.

3.10 IMPLEMENTATION OF POINTERS AND REFERENCES

- Pointers are mostly used in the heap-management. Same is true for references in Java. Due to these, pointers and references are not treated separately.
- Let us see the major problems with heap-management techniques.

3.10.1 Representations of Pointers and References

- In large computers, pointers and references use single values. Pointers stored in either two or four byte memory cells, depends on the size of address space of the machine.
- Segment and offset is used by Intel Microprocessors. So, pointers and references are implemented as pairs of 16 bit words, one for each of the two addresses.

3.10.2 Solution to dangling pointer problem

- There are several proposed solutions for dangling pointers:
  - Tombstone
  - Lock and Key

1. **Tombstone:**
   - Tombstone is an extra heap cell that is a pointer to the heap-dynamic variable.
   - Only at tombstones the actual pointer variable points.
   - Tombstone is set to nil, when heap-dynamic variable is de-allocated.
   - Tombstones are costly in Space and Time.

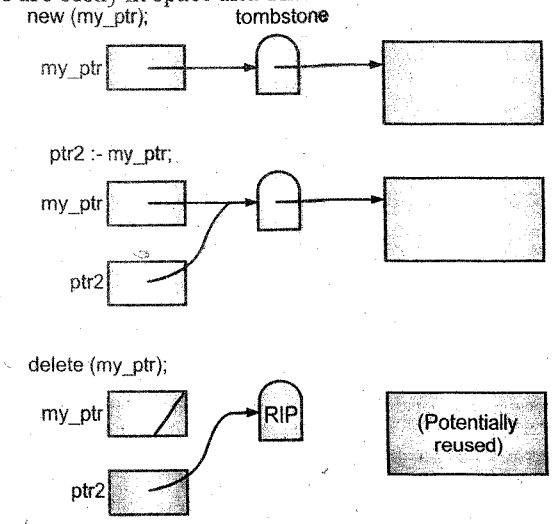


Fig. 3.17: Tombstone

2. **Locks-and-keys:**
   - Locks-and-keys use pointer values that are represented as (key, address) pairs.
   - For integer lock value, heap-dynamic variables are represented as variable + cell.

- Lock value is created and placed in lock cell and key cell of pointer, when heap-dynamic variable is allocated.

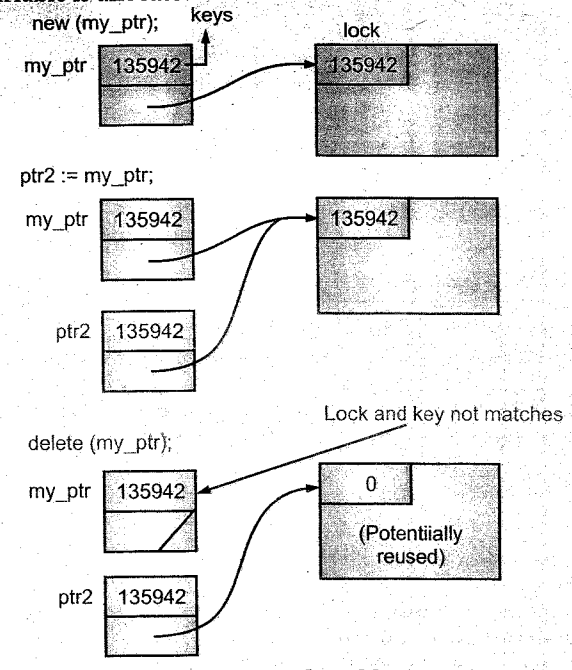


Fig. 3.18: Lock-and-keys

3.10.3 Heap Management

- Heap-management is a very complex run-time process.
- Single-size cells vs. variable-size cells.
- To reclaim garbage, there are two approaches:
  1. **Reference counters (eager approach):** Reclamation is incremental and is done when inaccessible cells are created (Gradual reclamation).
  2. **Mark-sweep (lazy approach):** Reclamation occurs when the list of variable space becomes empty.

3.10.3.1 Reference Counter

- Reference counter is the technique in which counter is maintained for each reference to variable.
- Counter start from zero and incremented according to number of pointers currently pointing at the cell.
- It is decremented when any reference is removed from cell.
- If it reaches zero, that means no program pointers are pointing at the cell, and therefore become garbage and can be returned to the list of available space.

Advantage:

- It is basically incremental, so significant delays in the application execution are avoided.

Disadvantages:

- Execution time required.
- Space required.
- Complications for cells connected circularly.

3.10.3.2 Mark-Sweep

- The run-time system allocates storage cells as requested and disconnects pointers from cells as necessary. Then Mark-Sweep process operates as follows:
  1. An extra bit in every heap cell is used by collection algorithm.
  2. Initially set to garbage all cells.
  3. Reachable cells are marked as not being garbage and all pointers are traced into heap.
  4. All garbage cells are returned to the list of available space.

Disadvantages of Mark-Sweep:

- It was done infrequently in its original form.
- It caused significant delays in application execution, when done.
- Contemporary mark-sweep algorithms avoid this by doing it more often called incremental mark-sweep.

3.10.3.3 Variable-Size Cells

- **Variable-Size Cells** avoids all the difficulties of managing single-size cells, but also has additional problems. These required by the most programming languages.
- If mark-sweep is used, additional problems occur as follows:
  - In the heap, the initial setting of the indicators of all cells is difficult.
  - The marking process is nontrivial.
  - Another source of overhead is maintaining the list of available space.

PRACTICE QUESTIONS

Q.I Multiple Choice Questions

1. Arrays in C++ are _____.
   (a) Row Major (b) Column Major
   (c) Diagonal Major (d) None of the above
2. A multi-dimensional array OPEN [1:3, 1:3, 2:3] contains _____ elements.
   (a) 20 (b) 18
   (c) 27 (d) 10

3. Which of the following data structure can't store the non-homogeneous data elements?
   (a) Arrays (b) Records
   (c) Pointers (d) None
4. Which are the correct array initialization statements?
   (a) int A[3]={1,2,3}; (b) int A[3]={123};
   (c) int A[3]="123"; (d) All
5. Which are the applications of array?
   (a) Sparse matrix (b) Ordered list
   (c) Both (a) & (b) (d) None
6. What is right way to initialize array?
   (a) int num[6] = { 2, 4, 12, 5, 45, 5 }; (b) int n{} = { 2, 4, 12, 5, 45, 5 };
   (c) int n{6} = { 2, 4, 12 }; (d) int n(6) = { 2, 4, 12, 5, 45, 5 };
7. What is the output of following code? #include <iostream>

```
using namespace std;
int main()
{ int I;
enum test{a=1,b,c,d,e,f,g,h};
for(i=c;i<=g;i++)
cout<<" "<<I;
return 0; }
```

   (a) Syntax error (b) 3 4 5 6 7
   (c) 1 2 3 4 5 (d) None of These
8. Explicit type conversion is known as _____.
   (a) Conversion (b) Casting
   (c) Separation (d) None of These
9. What is strong type system?
   (a) The type system in which only built-in data types are allowed.
   (b) The type system in which only user defined data types are allowed.
   (c) A type system that guarantees not to generate type errors.
   (d) None of these
10. A violation of type checking rules is known as _____.
   (a) Type cast (b) Type clash
   (c) Type inference (d) Type compatible
11. type Days is (mon, tue, wed, thu, fri, sat, sun);
   subtype Weekdays is Days range mon..fri;
   is example of _____.
   (a) subtype (b) subrange
   (c) enumeration (d) union

12. Type checking of unions require that each union include a type indicator called a _____.
   (a) free union (b) discriminant record
   (c) discriminant union (d) free record
13. A live pointer that no longer points to a valid object called _____.
   (a) pointer (b) reference
   (c) dangling reference (d) garbage
14. Which is not solution to dangling pointer?
   (a) Mark & sweep (b) Tombstone
   (c) Lock & keys (d) All are solutions
15. Which statement is incorrect?
   (a) Internal fragmentation happens when the method or process is larger than the memory.
   (b) External fragmentation occurs when memory is divided into variable size partitions based on the size of processes.
   (c) A referencing environment is the complete set of bindings in effect at a given point in a program.
   (d) Mark and Sweep is solution to fragmentation.

Answers

1. (a)	2. (b)	3. (a)	4. (a)	5. (c)	6. (a)	7. (b)	8. (b)	9. (c)	10. (b)
11. (b)	12. (c)	13. (c)	14. (a)	15. (d)					

Q.II Answer the following questions in short.

1. What are the different Primitive data Types?
2. What is a descriptor?
3. Define the term data type.
4. What are advantage and disadvantage of decimal data types?
5. List out the numeric data types.
6. What are the design issues in character string types?
7. Define three string length options.
8. Define ordinal type.
9. What are the design issues of enumeration type?
10. Define enumeration and subrange types.
11. What are the advantages of user defined enumeration types?
12. Explain Design issues of arrays.
13. Explain the different array types.
14. What are the advantages of each array type?
15. What is an aggregate constant?
16. Differentiate between slices of FORTRAN and ADA.
17. Explain Design issues of associative arrays.

18. Define the term Record.
19. Define row major order and column major order.
20. What is purpose of level numbers in COBOL records?
21. What are design issues of pointer types?
22. Define the term Union.
23. List out the problems with pointers.
24. Explain Design issues of Unions.
25. Describe lazy and eager approaches in reclaiming garbage.
26. Differentiate between C++ and java Reference variable.
27. What is dangling pointer?
28. What are the disadvantages of mark sweep?
29. Define reference counter.
30. What are the solutions for dangling pointers?

Q.III Answer the following questions.

1. Write a note on numeric data type.
2. Explain floating point data type.
3. What is Boolean type? Explain in details.
4. Explain any two character string length options in details.
5. Write a note on enumeration type.
6. Write a note on subrange type.
7. How the implementation of ordinal data is done?
8. Write a note on array types.
9. Explain the different categories of arrays.
10. What is the solution to dangling pointer problem? Explain in detail.
11. What is heap management? How it is done?
12. How the implementation of array is done?
13. Write a note on associative array.
14. How the implementation of record types is done?
15. Write a note on operations on records.
16. Explain the Discriminated versus free unions.
17. Write a different problem occurred in pointers.

■■■

CHAPTER 4

Control Flow

Contents

Objectives ...

After learning this chapter you will be able:

- To learn how to control flow changes using loops.
- To learn alternatives to use goto and jump statements.
- To do a comparative study of each control structure.
- To learn about short circuit conditions and their implementation.
- To design code with efficient control flow statement.
- To use recursion as an alternative to iteration.

4.1 INTRODUCTION

- We have already learned mechanisms that a compiler is used for checking semantic rules and the characteristics of the target machines for which compilers must generate code, we now return to core issues in language design.
- In this chapter, we learn types of control flow or ordering in program execution. According to the order of execution, tasks are performed.
- All languages use control statements than writing repeated code again and again, because the ability to write is enhanced by a larger number and wider variety of control statements.
- For example, instead of logically controlled loop statement for all loops, it is easier to write programs by a counter-controlled loop statement that controlled by a counter.
- A control structure is nothing but the collection of statements whose execution is controlled by a controlling statement.
- The eight principal categories of control flow are: Sequencing, Selection, Iteration, Recursion, Procedural abstraction, Concurrency, Non-determinacy and Co-lateral execution.

4.2 EXPRESSION EVALUATION

- Computations in functional programming languages are accomplished by evaluating expressions and functions.
- An expression consists of either a simple object or an operator or function applied to a collection of operands or parameters, each called expression.
- Order of evaluation may affect the result of the computation.
- In purely functional languages, an evaluation effect is the returned value only, with no side effects.
- Irrelevant of evaluation, order of sub-expressions is in purely functional languages.
- Operators are mostly placed in between operands.
- Operators do not need parenthesis.
- Consider the example of function : `F(a, b)`

Types:

- There are following types of expressions:

  Consider the expression `a+b-(c*d)`

  1. **Prefix (polish notation):** It is a notation in which operators precede their operands. The order is implicit.
     - Prefix conversion is: `-+ab(*cd)`

     Prefix with parenthesis called **Cambridge polish notation.** Example: `(-(+ab(*cd)))`
  2. **Infix:** It is a notation in which operators have resided between both the operands.
     - Infix conversion is: `(a+b-(c*d))`
  3. **Postfix (reverse polish notation):** It is a mathematical notation in which operators come after their operands. The order is implicit.
     - Postfix conversion is: `((ab+cd*)-)`

4.2.1 Precedence and Associativity

4.2.1.1 Operator Precedence

- Consider expression `a+b-(c*d)`. For this expression which operation will be performed first? a+b or c*d?
- In each language, the choice among alternative evaluation orders depends on the precedence and associativity of operators. Means preference is given to operators.
- For expression evaluation, **the operator precedence rules** define the order of adjacent operators of different precedence levels are evaluated. For example, multiplication and division group more tightly than addition and subtraction.

Table 4.1: Operator Precedence

Sr. No.	Operator	Description
1.	() [] ++ -- . ->	Parenthesis Pre increment, Pre decrement Indirection Operator
2.	++ -- + - ! ~ (type) * & sizeof	Post increment , Post decrement Bitwise Operators
3.	+ - / * % << >>	Binary Operators Left shift, Right shift
4.	< > <= >= == !=	Comparison Operators
5.	&& || ^ |	Logical Operators
6.	? :	Ternary Operator
7.	= += -= *= /= %= &= ^= <<= >>=	Shorthand Operators
8.	,	Sequencing

4.2.1.2 Operator Associativity

- Associativity rules specify whether sequences of operators of equal precedence follow rules for associativity from the right to the left.
- The basic arithmetic operators almost always associate from left-to-right, so 9-3+2 is 8 and not 4 (by 9-5).

Basic Associativity Rules:

- Associativity is from left to right, the exception is **FORTRAN** and **Ruby** which contains right to left.
- Sometimes unary operators associate right to left in FORTRAN.
- Equal precedence for all operators and right to left associated in APL.
- Associativity can be overridden with parentheses, precedence and associativity rules.

Table 4.2: Associativity with respect to each operator

Sr. No.	Operator	Description
1.	() [] ++ -- . ->	Left to Right
2.	++ -- + - ! ~ (type) * & sizeof	Right to Left
3.	+ - / * % << >>	Left to Right
4.	< > <= >= == !=	Left to Right
5.	&& || ^ |	Left to Right
6.	? :	Left to Right
7.	= += -= *= /= %= &= ^= <<= >>=	Right to Left
8.	,	Left to Right

4.2.2 Assignment

- Effect of computation of expression is assigned to resulting variable using the assignment.
- The syntax in general:

```
<target-var><assignment-operator><expression>
```

- In FORTRAN, BASIC, the C-based languages, assignment is represented by the '=' operator.
- In ALGOL, Pascal, ADA languages, assignment is represented by the ':=' operator.

1. **Conditional Targets (PERL) with assignment:**

- Assignment operator used for conditional targets shown in the following example:

```
($flag ? $T: $S) = 1
```

- This is equivalent to:

```
if ($flag)
{  $T = 1 }
else
{  $S = 0 }
```

2. **Compound Assignment Operators:**

- It specifies a generally needed form of assignment.
- It is introduced in ALGOL and adopted it in C.
- For example, 'p =p + q' is written as 'p += q'

3. **Unary Assignment Operators:**

- In C-based languages, with assignment we can use a combination of increment and also decrement operations.
- For example,

```
total + = ++i ( i is incremented, added to total)
total + = i++ ( i is incremented, added to total)
p++ ( p incremented)
-p++ (p incremented then negated)
```

4. **Assignment as an Expression:**

- The assignment statement produces a result that can be used as operands in C, C++, and Java.

```
while ((ch = getchar())!= EOF)
        {...}
```

- Here, `ch = getchar()` is carried out; the result (assigned to ch) is used as a conditional value for the while statement.

5. **List Assignments:**

- List assignments are supported by Perl and Ruby.

Example: `($Ones, $Tens, $Hundreds) = (1,10, 100);`

6. **Mixed-Mode Assignment:**

- Assignment statements can also be mixed-mode.
- For example,

```
int x, y;
float z;
z = x / y;
```

- Numeric type value can be assigned to numeric type variable, in Fortran, C, and C++.
- Widening assignment coercions are done only in Java.
- No assignment coercion in Ada language.

7. **Assignment in Imperative Programming:**

- Imperative Programming is also called "programming with side effects".
- Few Languages (Euclid and Turing) do not allow side effects in a function that returns a value. If repeatedly called with the same set of arguments, it will always produce the same answer.
- Imperative languages use assignment.
- **Computation** is a series of changes to variable's values in memory in imperative languages.
- In imperative languages, the outcome of the computation may be determined by the order in which these side effects happen.
- There is usually a distinction between an expression and a statement.

Side Effects:

- A function has a side effect of computation is done other than by returning a value. When a function changes the environment in which it exists, a side effect occurs.
- In an imperative language, computation consists of an ordered series of changes to the values of variables in memory. Assignments provide the principal means by which to make the changes i.e. called a **side effect**. For example, `A=2;`
- Many (though not all) imperative languages which always produce a value, and may or may not have side effects.
- Sometimes side effects are desired. Consider the following example,

```
int rand();
```

It needs a side effect, otherwise every time it is calling the same random number.

- Check **evaluation of Operands and Side Effects**. Consider the code given below,

```
int x = 0;
int foo()
{
   x += 5;
   return x;
}
int a = foo() + x + foo();
```

In above code, what is the value of a?

Idempotent:

- The function if called again with the same parameters, it will always give the same result is called an Idempotent Function.

8. **Assignment in Declarative Programming:**

Special cases of assignment:

- It not contains assignments i.e. no side effect but only uses functions. For example, purely functional languages have no side effects.
- The value of an expression in this type of language depends only on the referencing environment where the expression is evaluated, not at the time of evaluation.

For example,

```
+ab          //declarative
c=a+b        //imperative
```

- **Referentially Transparent Expression**: If an expression yields a certain value at one point in time, it is guaranteed to give the same value at any point in time.
- Expressions are referentially transparent in a purely functional language, as their value depends on referencing environment only. For example, Haskell and Miranda.
- Consider the following C code:

```
i=3;
sizeof(i++) or sizeof(i--)   //2 byte
print i;                     //3
```

9. **Combination Assignment:**

- As some languages deeply trust on side effects, imperative programs must frequently update variables.
- Consider statement, `B[i++]=++j;`
- This statement has following series of assignments as follows,

```
j=j+1;
B[i]=j;
B[i]=B[i+1];
```

- When we use the prefixed expression, the ++ or -- operator increments or decrements its operand before providing a value to the surrounding context. In the postfix form, ++ or -- updates its operand after assigning a value.
- If i = 1 and j=2, then j will be assigned into array's location B[1] (not B[2]), and the next assignment will copied to B[2].

10. **Multiway Assignment:**

- We can do many variable assignments at the same time.

(i) Swapping:

- First use of multiway assignment is **swapping**.
- We have already seen the right associativity of assignment, we can use a=b=c. In simple multiway assignment of several languages, like CLU, ML, Perl, Python, and Ruby, it is also possible to write,

```
a,b=b,a
```

- Alternative way for the above assignment in C is,

```
temp=a;
a=b;
b=temp;
```

Example:

```
a=10, b=30, c=20
a, b, c = b, c, a
```

So here,

```
a=b
b=c
c=a
```

(ii) Return multiple values from subroutine:

- Consider the code below,

```
a, b, c=F()
F()
{
    ..
    return (x,y,z);          //at a time, 3 values returned from F()
}
```

- Means final assignment is like,

```
a=x
b=y
c=z
```

4.2.3 Initialization

- In initialization, we use the assignment statement to set the value of a variable. Imperative languages do not provide an initial value for a variable in its declaration.
- There are following three reasons where initialization is required:
  1. A static variable that is local to a subroutine requires an initial value in order to be useful.
  2. For statically allocated variables, an initial value given at declaration can be allocated previously by the compiler in global memory, to avoid the cost of assignment at run time. That means allocation should be in the static region, otherwise will get mismatch/incorrect result.
  3. Most languages give errors if no initialization done for variables i.e. only declaration not supported by some languages.
- To prevent such errors, it is necessary to give every variable a value when it is first declared.
- If there is no initial value explicitly provided to variable in its declaration, some languages may assign a default value. For example, in C, statically allocated variables by default are initialized to zero.

Definite Assignment:

- Java and C# uses definite assignment that prevents the use of uninitialized variables.
- For example, variable declaration in Java.

```
class A
{
    public static void main(args[])
    {
    int i;
        //when variable i declared in main as no value assigned it
        contains garbage
    ..........
    System.out.println(i);
    }
}
```

- Now consider the alternative code,

```
class A
{
    int i;
    public static void main(args[])
    {
            //when i declared before main though no value assigned it
            contains 0 by the default value of the integer.
    ..........
    System.out.println(i);          //0
    }
}
```

- Aggregates are compile-time constant values that mean values are initialized at compile time.

4.2.4 Ordering within Expression

- While precedence and associativity rules define the order, the binary infix operators are applied within an expression.
- If we do not specify the order in which the operands of a given operator are evaluated in any order.

Example 1:

```
a-f(b)-c*d
```

In this example, the result of the expression depends on returned values from f(b). If f(b) makes updates to any variable a, c or d.

```
f(b)
{
    c++;
    d--;
    return c;
}
```

Example 2:

```
f(a,g(b),c)
```

In this example, g(b) makes updates to any variable a or c.

```
g(b)
{
   a++;
   c--;
}
```

If the order of evaluation changes then we will get the different result every time.

Tracing of the Output:

```
main()
{
   int i=3;
   printf(i,i++,++i,i)
}
```

Output:

```
i, i++, ++i, i
5  4    4    3
```

Importance of Order:

- There are the following two main reasons why the order is important.

1. **Side Effects:**

- Assignment or Changes done to variables then it makes side effects.
  As we have seen above in the 1st expression,
  If `f(b)` may modify d, then the value of `a-f(b)-c*d` will depend on whether subtraction or multiplication is performed first.
- Similarly in the second example, if `g(b)` modifies a or c, then the values passed to `f(a, g(b), h(c))` will depend on the order in which the arguments are evaluated.

2. **Code Improvement:**

- The order of evaluation of sub-expressions has an impact on both register allocation.
- In the expression `a * b + f(c)`, it is needed to call `f` before evaluating `a*b`, because the product, if calculated first, would need to be saved during the call to `f`, and `f` may use registers to save the value.
- Also compiler not makes recalculations.

  Example 1:

```
C=a*b
D=d/a*b
```

- As `a*b` is required again while computing D, so it stores computation result in the register rather again calculations.

  Example 2:

```
a:= B[i];
c:= a * 2 + d * 3;
a:= B[i];
```


- Requires to load and store value in memory. Instructions array may require more time for multiplication `d*3` calculated first then `a*2` calculated in C.
- Similarly,

```
a = b/c/d
e = f/d/c
```

May be rearranged as,

```
t=c*d  //as both expressions need this same value.
a = b/t
e = f/t
```

- Evaluation order impacts register allocation, instruction scheduling, etc.
- Though we make a particular evaluation ordering, some code improvements can not be possible. This impacts performance.

4.2.5 Short-circuit Evaluation

- An expression in which without evaluating all of the operands and/or operators the result is determined.
- Consider the expression, `(25*a)*(b/9-2)`. If 'a' is zero, there is no need to evaluate `(b/9-2)`.
- Boolean expressions make improvements in code and also readability increases.
- Consider the expression, `(a < b) and (b < c)`. Here, both expressions need to be true as we use and.
- Now consider, `(a < b) or (b < c)`. Here, either one of the expressions needs to be true as we use or. So consider, if the first expression returns true it doesn't check the second expression as or used. So the second part is skipped, called **short-circuit evaluation**.
- Consider the following code,

```
if(a&&b||c)
int a=4,b=3,c=5;
if(++a && --b||--c)
printf(a+b+c)
```

- Short-circuiting is not necessarily as attractive for situations in which a Boolean expression which cause a side effect. In some languages like Pascal for counting expressions, if short-circuit evaluation is used, the program will not compute the right result.
- The code can be rewritten by a non–short-circuit evaluation. So for the solution of some languages provide both regular and short-circuit Boolean operators.
- For example **in Ada**,

```
found_it:= p /= null and then p.key = val;
// Ada uses /= for "not equal."
...
if d = 0 or else n/d < threshold then …
```

- In Ada, delayed or lazy evaluation takes place that means the operands are "passed" unevaluated or as it is.
- In C, the bitwise & and | operators are used as non–short-circuiting alternatives to && and || when their arguments are logical (0 or 1) values.
- For example,

```
i=1;
while(i<= limit)&&(A[i] != value)
    i++;
```

- Here, A[i] will cause an indexing problem when i=limit, considering that A has limit -1 elements.

4.3 STRUCTURED VS. UNSTRUCTURED FLOW

4.3.1 Introduction

- Control Flow in Assembly languages achieved using unconditional jump statement (branches).
- Early versions of Fortran relying heavily on goto statements for most non-procedural control flow:

```
if (A.lt.B) goto 10        // ".lt." means "<"
...
10:
```

- Here, 10 at the bottom is the label. Goto statements also featured prominently in other early imperative languages.

Features of Structured Programming:

- The rejection of goto statements was part of a larger "revolution" in software engineering known as Structured Programming.
- It is a popular programming trend in the 1970s. It uses a Top-down design.
- It also uses modularization of code, structured types like records, sets, pointers & so on.
- It uses descriptive variable names and extensive commenting after Algol 60, most languages had: `if...then...else, while` loops. So they don't need to use goto statements.

4.3.2 Common use of goto and Structural Alternatives to goto

Following are uses of goto statement:

1. **Mid loop exit and continue:**

- Here, code breaks early in mid of loop. As we have taken label at the end of loop goto statement will terminate execution of loop early.

Use of goto	Structural alternative

for(){
…
..
 	
goto L1;
..
..
}
L1:…
	for()
	{
	…
	..
	break;
	..
	..
	}
	…

- For example, Pascal.

2. **Early return from subroutine:**

- Before finishing the body of subroutine, it returns early to called function as we used goto in the conditional loop. If the condition is satisfied goto statement then it will terminate the execution of the loop early.

Use of goto	Structural alternative
P() Q() { { … …. Q(); if(a<b) …. goto L1 L1:… ….. } }	P() Q() { { … …. Q(); if(a<b) …. return; … ….. } }

3. **Errors and other exceptions:**

- In a common situation, a deeply nested block or subroutine may discover that it is unable to proceed with its function, and there is a lacks the contextual information it would need to recover in any way.
- Eiffel formalizes this notion by saying that each software component has a contract - a specification of the function it performs.
- A component that is not able to fulfill this type of contract means if the error comes is said to fail. Rather than return in the normal way, control comes "back out" to some context in which the program is able to recover.

- Conditions that require a program to "back out" are called exceptions.

Use of goto	Structural alternative
{	{
….	….
..	..
if(error)	if(error)
goto L1;	exit(1);
….	….
…	…
}	}
L1:…	…

4. **Multilevel return:**

- Return and goto statements allow control to be returned from the current subroutine. Now we want to return from a surrounding routine.
- Consider nested subroutine example, the search routine might invoke several nested routines, or a single routine multiple times, once for each place in which to search.
- In such situation certain historic languages, including Algol 60, PL/I, and Pascal, permit a goto to branch to a lexically visible label outside the current subroutine:

Use of goto	C alternative	Java
for (){	for (){	Java:
…	…	for (){
..	..	…
for (){	for (){	..
{	{	for (){
…	…	{
while(condition)	while(condition)	…
{	{	while(condition)
…	…	{
for()	for()	…
{…	{…	For()
goto L1;	break;	{…
..	..	break L1;
..	..	..
}	}break;	..
}	}	}
}	break;	}
}	}	}
L1:….	break;	}
	}	L1:….
	…….	

4.3.3 Continuation

- The notion of non-local goto statements that unwind the stack known as Continuations.
- In low-level terms, Continuation consists of code address and referencing environment to be restored when jumping to the particular address whereas in higher-level terms, a Continuation is an abstraction that captures a context in which execution may continue.
- First-class Continuations are extremely powerful facilities. They are very useful if applied in well-structured ways. Unfortunately, they also allow undisciplined programmer to design completely inscrutable programs.

Types of Control flow:

- As we have seen earlier, there are eight types of control flow:
  1. Sequencing
  2. Selection
  3. Iteration
  4. Procedural Abstraction
  5. Recursion
  6. Concurrency
  7. Non-determinacy
  8. Collateral Execution
- Now, will learn one by one in brief.

4.4 SEQUENCING

- Same as an assignment, sequencing is also used in imperative programming. It means controlling the order in which side effects occur.
- When one statement occurs before another in the program text, the first statement executes before the second.
- Sequencing comes naturally without any special syntax to support it in imperative languages.
- For sequencing provide special constructs in mixed imperative/function languages (For Example, Scheme, LISP).
- The issue is that what is the value of a sequence of expressions/statements?
- Most common is the value of the last sub-expression.
  - In LISP,

    ```
    (prog1 (setq x 10) (setq y 20))  =>20
    ```

  - In C,

    ```
    a = 10, b = 20;                  =>20
    ```

- The value of the first sub-expression in LISP,

  ```
  (prog1 setq x 10) (setq y 20))=>10
  (prog1 setq x 10) (setq y 20) (setq z 30)) => 10
  ```

- The value of the second sub-expression in LISP:

```
(prog1 setq x 10) (setq y 20)) => 20
(prog1 setq x 10) (setq y 20) (setq z 30)) => 20
```

- In most imperative languages, lists of statements can be enclosed within begin... end pair or {... } braces. Such delimited list of statements is called a Compound Statement.
- A Compound Statement optionally preceded by a set of declarations is sometimes called a Block. This means it groups multiple statements together.
- **Basic block** (This will be discussed later after compilers): Block in which only sequencing is allowed as control flow.

Examples:

- { } in C, C++, and Java.
- `Begin/end` in Algol, Pascal, and Modula.

4.5 SELECTION

4.5.1 Basic Concept

- A selection statement provides the means of choosing between two or more execution paths in a program.
- There are two general categories:
  (a) Two-way Selectors: Choosing one of two execution paths in a program.
  (b) Multiple-way Selectors: Choosing one of any number of (Multiple) execution paths in a program.

1. **Two-way Selectors:**

- `if... then... else` statements used to achieve selection called Two-Way Selection Statements.

```
if(condition) then
    statement
else
    statement
```

- Nested if statements (Nesting Selectors) have a dangling else problem in some languages.

```
if(condition) then
        // statement
else if( condition) then
        // statement
...
else
        //statement
```


- There are differences to use else statement in various languages:
  - **Algol 60**: This language does not allow the "then if" statement.
  - **Pascal**: In this language, `else` associates with closest unmatched then statement.
  - **Perl**: This language has a separate `elseif` keyword (in addition to else and if). The statement "elseif" will cause an error.
  - **LISP**: Consider cond function in LISP,

```
(cond ((= A B) (...)) ((= A C) (...)) ((= A D) (...)) (T (...))
```

  is the same as,

```
if -else if- else series.
```

2. **Multiple-way Selection:**

- Example in C, C++, and Java is:

```
switch(expression)
{
    case Expr1: Stmt1;
    ...
    case ExprN: StmtN;
    [default: StmtN+1]
}
```

- In C, the control expression can only be an integer type.
- Blocks, Statement sequences, or Compound statements can be selectable segments.
- In one execution of the construct, a number of segments can be executed.
- 'Default' clause is used when not any case matches the value. If no default case then for the values which are not represented nothing will be done.
- In Ada, the syntax is:

```
case expression is
    when ch_list1 => stmt-seq1;
        ...
    when ch_listN =>stmt-seqN;
    when others =>stmt-seq;
end case;
```

4.5.2 Short-circuited Conditions

- In some languages, it is necessary to evaluate all operands before applying operators called **strict evaluation**. For example, Pascal.
- In Short-circuit Evaluation, we can skip operand evaluation when possible. For example, when *or* operand is used.
  - In C++ and Java, || and && always use short-circuit evaluation.
  - In Ada language, `then if` and `or else` supports both strict and short-circuit. The programmer decides: use of *and*, *or* for strict evaluation.
  - In Perl language, `Open (FILE, ">filename.ext") or die ("error!");`

- We can avoid out-of-bounds/dereferencing NULL reference errors, or divide by zero. So it can lead to more efficient code. Most of the time short-circuits can be problematic when conditions have side effects.

4.5.3 Case / Switch Statements Iteration

- In Algol-W and its descendants, we have an alternative to nested if...then...else blocks.
- Consider the following example:

```
cnt:= … (* potentially complicated expression *)
IF cnt = 1 THEN clause_1
ELSEIF cnt IN 2,7 THEN clause_2
ELSEIF cnt IN 3..5 THEN clause_3
ELSEIF (cnt = 10) THEN clause_4
ELSE clause_5
END
```

- Same code can be rewritten as,

```
Case … (* potentailly complicated expression *) of
   1 :    clause_1
   2, 7:  clause_2
   3..5:  clause_3
   10:    clause_4
   ELSE   clause_5
END
```

- The constants in the label, in cases must be disjoint and a type compatible with the tested expression. Most languages allow this type to be anything whose values are discrete like integers, characters, enumerations, and subranges of the same. C# also allows strings.
- Implementation of Case Statements or Nested If...then...else can be done as,

```
        r1 := …
        if r1 <> 1 goto L1
        clause_1
        goto L6
L1 :    if r1 = 2 goto L2
        if r1 <> 7 goto L3
L2 :    clause_2
        goto L6
L3 :    if r1 < 3 goto L4
        if r1 > 5 goto L4
        clause_3
        goto L6
L4 :    if r1 <> 10 got L5
        clause_4
        goto L6
L5 :    clause_5
L6 :
```


- Rather above code, we can use a jump table.

```
T  :  &L1          -- tested expression = 1
      &L2
      &L3
      &L3
      &L3
      &L5
      &L2
      &L5
      &L5
      &L4                 -- tested expression = 10
L6 :  r1 :=               -- calculate tested expression
      if r1 < 1 goto L5
      if r1 > 1 goto L5   -- L5 is the "else" arm
      r1 -:= 1
      r2 := T[r1]
      goto *r2
L7 :
```

- Here the "code" at label T is actually a table of addresses, which is called a **jump table**. It contains one entry for each integer between the lowest and highest range values, inclusive, found among the case statement labels.
- By considering *case* versus *switch*, we can see alternative C switch code for above code,

```
switch (... /* tested expression */)
{   case 1: clause_1
       break;
    case 2:
    case 7: clause_2
       break;
    case 3:
    case 4:
    case 5: clause_3
       break;
    case 10: clause_4
       break;
    default: clause_5
       break;
}
```

- This type of Switch statement is also used in C++, and Java which have unique syntax. It uses break statements otherwise statements fall through to the next case.
- Case is used in most other languages. We can have ranges and lists in the switch. Some languages such as Pascal do not have default clauses or cases.
- In some languages like FORTRAN, Case statements descended from the computed goto statement.

Example:

```
goto (15, 100, 150, 200), J
If J is 1, then it jumps to label 15
If J is 4, then it jumps to label 200
If J is not 1, 2, 3, or 4, then the statement does nothing
```

- Consider the following *if* code for short-circuit evaluation is given and we have to write jump code implementation.

```
if((A>B)&(C>D) or(E<>F)) then
   then_clause
else
   else_clause
```

- **Jump code implementation** is:

```
r1=A
r2=B
r1=r1 > r2
r2=C
r3=D
r2=r2 > r3
r1=r1 & r2
r2=E
r3=F
r2=r2 <> r3
r2=r1 or r2
if r1=0 and r2=0
   goto L1
else
   goto l2
L2:then clause
L1:else_clause
```

- Consider Ruby's case expression statement,

```
case
when boolean_expression the expression
...
when boolean_expression the expression
 [else expression]
end
```


- The value of the *case* expression is executed first then expression whose boolean expression is true. In this statement, the *else* represents true, and the *else* clause is optional.

4.6 ITERATION

- An iterative statement is one that causes a statement or collection of statements to be executed more than one time. An iterative statement is also referred to as a Loop.
- Iteration and recursion are the two mechanisms that allow a computer to perform the same operations again and again.
- Running time of the program will be dependent on a number of iterations. This means it is iteration and recursion that make computers useful. At this point, we focus on iteration.
- Iteration is a more prevalent concept in imperative languages which can be carried out in the form of loops.
- Iteration of loops used to make side effects. So repeatedly modification of variables takes place.
- Iterative loops come in two principal varieties:
  1. Enumeration-Controlled Loop
  2. Logically Controlled Loop.

4.6.1 Enumeration-Controlled Loops

- This loop executed once for every value in a finite set. Enumeration-controlled iteration is first used as, 'do loop' in Fortran.
- Consider the following code,

```
do i = 1, 50, 2
   ...
enddo
```

Here, Variable i is the counter or index of the loop. The body of the loop between the loop header and the *enddo* delimiter will execute 25 times, with i set to 1, 3,... , 9 in successive iterations.

- Equivalent code with *label* for above code is,

```
L1: i = 1
    ...
i = i + 2
if i<= 50 goto L1
```

Semantic Complication:

- Following issues may arrive:
  - **Issue 1**: Can step size/bounds be Positive or negative?
  - **Issue 2**: Changes to loop indices or bounds. Algol 68, Pascal, Ada, Fortran 77/90, exclude changes to the index within the loop and evaluate bound once before iteration.

- **Issue 3**: Test terminating condition before the first iteration.
  Example:

```
for i:= first to last by step do
    ...
end
```

  - jump code implementation of the above code is,

```
r1:= first
r2:= step
r3:= last
L1: if r1 > r3 goto L2
 ...
   r1:= r1 + r2
   goto L1
L2:
```

  - Alternatively we can write,

```
r1:= first
r2:= step
r3:= last
L1: ...
r1:= r1 + r2
L2: if r1 <= r3 goto L1
```

- **Issue 4**: Restrictions on entering loop from outside.
  - Algol 60 and Fortran77 and most of their descendants prevent the use of goto's to jump into a loop. Mostly "exit" or "continue" is used for loop escape.

4.6.2 Iterators

- This method allows a computer to perform the same information repeatedly.
- For example, for loop iterates over the elements of an arithmetic sequence.
- In general, however, we can iterate over the elements of any well-defined set such as collections & so on.
- For solution, Clu introduced an iterator mechanism which also found in Python, Ruby, and C#. These languages allow any container abstraction to provide an iterator that enumerates its items is called **True Iterators**.
- Euclid and several more recent languages, especially C++ and Java, define a standard interface for iterator objects which are also called **Enumerators**. These are very easy to write.

4.6.3 Logically Controlled Loops

- By comparing with enumeration-controlled loops, logically controlled loops have their own different semantic meaning common approach is to test the condition before every iteration. For example, the familiar 'while' loop in Pascal.
- Consider while loop in Algol-W:

```
while condition do
statement
```

- This loop executes until some boolean condition is raised, depends on the value altered in the loop.
- We all know the advantages of **for loop** over **while loop.**

Advantages of Logically Controlled Loop:

- Here are some Advantages of Logically Controlled Loop:
  - Loops having compact nature.
  - The clarity is very good.
  - All code affecting flow control is localized in the header.

Design issues:

- There are some design issues of:
  - Should the control be pre-test or post-test?
  - Should the logically controlled loop be a special form of a counting loop or a separate statement?

Types of Logically controlled loop:

- There are 3 types according to the place to test termination condition:
  1. **Post-test loops:**
     - It is a type of loop where a terminating condition presents at the bottom of a loop.
     - Pascal introduced special syntax for this loop, which was retained in Modula but dropped in Ada.
     - Examples of post-test loops are: do-while, repeat - until loop.
     - **Example**:

```
repeat
readln(line)
until line[1] = '$';
```

     - We can convert the same code to 'do-while' loop in C.

```
do
{
line = read_line(stdin);
}
while (line[0] != '$');
```

     - In C, in do-while loop, condition is checked after the block is executed.

2. **Pre-test loop:**
   - Logically-controlled pre-test loops check the exit condition before the next loop iteration.
   - For example, in C, C++, the while loop terminates when the condition evaluates to 0, NULL, or false. It uses to continue and break statements to jump to the next iteration or exit the loop.
   - In Java, the condition is restricted to Boolean.
3. **Mid-test for loop:**
   - In some loops, terminating condition present in the middle of a loop. In many languages this "mid test" can be accomplished with a special statement nested inside a conditional: `exit` in Ada, `break` in C, `last` in Perl.
   - We know use of break to leave a C switch statement. C also uses break to exit from the `for`, `while` or `do` loop.
   - Consider the given code in C that uses logically controlled loop:

```
for(i = first; i<= last; i += step)
{
   ...
   if (i>2) break;
}
```

   - Alternative way to use while loop,

```
i = first;
while(i<= last)
{
   ...
   if(i>2) break;
   i += step;
}
```

4.6.4 Combination Loops

- Some languages allow modern enumeration loop means combination of enumeration and logically controlled loops.
- For example, Algol 60's `for` loop.

```
for i:= 1, 3, 7, 9 do…
for i:= 1 to 9 by step do…
END
for i:= 1 step 2 until 10 do …
for i:= 1, i + 2 while i < 10 do …
for (i = first; i<= last; i += step)
{... }
```


- C defines this loop equivalent to while loop,

```
{
   i = first;
   while(i<= last)
   {
      ...
      i += step;
   }
}
```

- There are few problems with combination loops such as,
  - Repeated evaluation of bounds may occur.
  - It is very hard to understand.

4.7 RECURSION

4.7.1 Basic Concept

- Recursive Computation can be done as, "Decompose the problem into smaller problems by calling itself again and again with the same name."
- **Base case:** When the function does not call itself any longer then there will be no base case and no return value.
- Problem must always get smaller and approach to the base case.
- There are no side effects using recursion. It requires no special syntax.
- It can be implemented in most programming languages that need to permit functions to call themselves (direct recursion) or other functions that call them in return (indirect recursion).
- Some languages such as Fortran 77 do not permit recursion.
- Consider the following tracing of Recursive Function `sum()`:

```
(define sum (lambda(n)
(if (= n 0)
0
(+ n (sum (- n 1))))))
>(trace sum)
#<unspecified>>
>(sum 5)
sum 5
   sum 4
      sum 3
```

```
            sum 2
               sum 1
                  sum 0
                  sum 0
               sum 1
            sum 3
         sum 6
      sum 10
   sum 15
```

so, output=15.

4.7.2 Iteration Vs. Recursion

- **Iteration**: It is repeated modification of variables or makes side effects. Mostly used loop in imperative languages. It uses a repetition structure (Examples: for, while loops). It terminates when the loop continuation condition fails.
- **Recursion**: It does not change variables or have no side effects. Mostly used loop in functional languages. It uses a selection structure (Examples: if, if-else, or switch-case). It terminates execution when recognized as a base case.
- Any logically controlled iterative algorithm can be rewritten using a recursive algorithm and vice versa.
- Iterative procedure for while loop:

```
while (Expr)
{
   Stmt1; Stmt2;
... }
```

- This can be turned into a recursive one as shown in the following code.

```
procedure P()
{
   if (Expr)
   {
      Stmt1; Stmt2;
      ...; P();
   }
}
```

- The converse is not true (e.g., Merge Sort, Quick sort, matrix multiplication,...)
- The type of recursive procedure above can be translated back into a loop by the compiler called **tail recursion**.

- Consider the following two examples of recursion & guess, which is tail- recursive?

Example 1:

```
(define summation (lambda (f low high)
(if (= low high)
(f low)
(+ (f low) (summation f (+ low 1) high)))))
```

Example 2:

```
(define summation (lambda (f low high subtotal)
(if (= low high)
(+ subtotal (f low))
(summation f (+ low 1) high (+ subtotal (f low))))))
```

4.7.3 Tail Recursion

- If recursive call statement is present as the last statement of function then it is called Tail Recursion.
- It is sometimes called that iteration is more efficient than recursion. The iterative implementations of summation example above will be more efficient than the recursive implementations, if later on we make real subroutine calls, space will be allocated on a run-time stack for local variables and book-keeping information.
- An "optimizing" compiler, for a functional language we can generate excellent code for recursive functions.
- Compared to embedded recursion Tail Recursion is better as additional computation never follows a recursive call. The return value is whatever the recursive call returns only.
- The compiler can reuse space that belongs to the current iteration when a recursive call is made. Though there is unnecessary dynamically allocated stack space.

Example: Consider the recursion code below and for that, we have to remove Tail Recursion.

```
int gcd(int a, int b)
{
   if(a==b) return a;
   else if (a > b)
      return gcd(a-b, b);
   else
      return gcd(a, b-a)
}
```

- Now, after removing the Tail Recursion code looks like,

```
int gcd(int a, int b)
{
start:
   if(a==b) return a;
   else if(a > b)
   {
      a = a-b;
      goto start;
   }
   else
   {
      b = b-a;
      goto start;
   }
}
```

4.7.4 Applicative and Normal-Order Evaluation

- It is assumed that arguments are evaluated before passing them to a subroutine. But sometimes, it is possible to pass unevaluated arguments to the subroutine and to evaluate them only when (if) the value is actually needed means on-demand.
- According to evaluation time, there are two categories:

1. **Applicative-order Evaluation:**
   - Evaluating arguments before the call is known as Applicative-order evaluation.
   - For clarity and efficiency, applicative-order evaluation is generally preferable than normal-order evaluation. It is therefore natural for it to be employed in most languages.
   - For example, Subroutine calls or Function calls.
2. **Normal-order Evaluation:**
   - The latter evaluating arguments means only when the value is actually needed is known as Normal-order evaluation. Normal-order evaluation naturally occurs in macros.
   - It also occurs in short-circuit Boolean evaluation, call-by-name parameters and also in certain functional languages.
   - For example, in Algol 60 we use normal-order evaluation by default for user-defined functions but sometimes applicative order can also be possible.
   - Mainly most programmers in 1960 wrote in assembler, and were familiar with macro facilities. Side effects become an issue.

3. **Lazy Evaluation:**
   - As we know that applicative order is most preferred. In some circumstances, however, normal-order evaluation can generate faster code, or generate error-free code, when applicative order evaluation would lead to a run-time error.
   - Difference is normal-order evaluation will sometimes not evaluate an argument at all, if its value is never actually needed.
   - Scheme provides for optional normal-order evaluation in the form of built-in functions called delay and force called lazy evaluation.
   - Promise is nothing but a delayed expression. The mechanism used to keep track of promises to be evaluated is sometimes called Memorization.

Example: Consider the following code and its evaluation with both techniques.

```
(define double (lambda (x) (+ x x)))
```

Applicative-order/AO/subroutine:

```
(double(* 3 4))      //parameters evaluated then passed
```

- (double 12)
- (+ 12 12)
- 24

Normal-order/NO/macro/lazy evaluation:

```
(double(* 3 4))      //parameters passed then evaluated
```

- (+ (* 3 4) (* 3 4))
- (+ 12 (* 3 4))
- (+ 12 12)
- 24

SUMMARY

- Expression Evaluation can be done in three ways: Prefix, Postfix and Infix.
- Imperative programming is also called "programming with side effects".
- A function has a side effect if computation is done other than by returning a value.
- An Idempotent function is one that of called again and again with the same number of parameters, will always give the same result.
- If an expression yields a certain value at one point in time, it is guaranteed to yield the same value at any point in time called referentially transparent.
- Multiway assignment can be used for swapping or returning multiple values at once.
- Orthogonality is one of the principal design goals to combine various features of the language; with the combinations all make sense or are meaningful.

- Java and C# uses definite assignment that precludes the use of uninitialized variables.
- Common 4 uses of goto statement are Mid-loop exit & continue, Early return from subroutine, Errors and other exceptions and Multilevel return.
- Types of control flow:
  1. **Sequencing:** When one statement occurs before another in the program text the first statement executes before the second.
  2. **Selection:** A selection statement provides the means of choosing between two or more execution paths in a program. For example, if-else, switch.
  3. **Iteration:** An iterative statement is often called a loop.
     - Iterative loops come in two principal varieties: Enumeration-Controlled Loop, Logically Controlled Loops.
     - Some languages allow modern enumeration loop means Combination of enumeration and logically controlled loops called Combination Loops.
  4. **Procedural abstraction:**
     - **Recursion:**
       - If recursive call statement is present as last statement of function it is called Tail Recursion.
       - We can write code by removing tail recursion means just by using goto statement.
     - **Evaluation have two categories:**
       (i) Applicative (Evaluating arguments before the call to function is known as applicative-order evaluation) and
       (ii) Normal-Order Evaluation (The latter evaluating arguments means only when the value is actually needed is known as normal order evaluation). Scheme provides for optional normal-order evaluation in the form of built-in functions called delay and force called lazy evaluation.

PRACTICE QUESTIONS

Q.I Multiple Choice Questions.

1. When a function calls itself, it is known as _____.
   (a) Self Referential (b) Recursion
   (c) Repeated Call (d) Loop
2. Indirect recursion is also called as _____.
   (a) Mutual Recursion (b) Redirect Recursion
   (c) Tail Recursion (d) None of the above
3. Mutual recursion is also called as _____.
   (a) Indirect Recursion (b) Redirect Recursion
   (c) Tail Recursion (d) None of the above

4. A recursive function is said to be _____ recursive if there are no pending operations to be performed on return from a recursive call.
   (a) End (b) Tail
   (c) Linear (d) Binary
5. The simplest form of Recursion is _____.
   (a) Linear Recursion (b) Indirect Recursion
   (c) Mutual Recursion (d) Tail Recursion
6.
```
fact(n)
begin
   if(n<=1) return 1;
   else return n*fact(n-1)
end
```
   In the above code _____.
   (a) Function is example of Direct Recursion
   (b) Function is example of Indirect Recursion
   (c) Function is example of Mutual Recursion
   (d) Both (a) and (b)
7. When recursive call is the last statement in the function, it is called _____.
   (a) End recursion (b) Tail Recursion
   (c) Base Case (d) Indirect Recursion
8. What is the output of the following C code?
```
#include <stdio.h>
void main()
{
   int a[2][3] = {1, 2, 3, 4, 5};
   int i = 0, j = 0;
   for(i = 0; i< 2; i++)
   for(j = 0; j < 3; j++)
   printf("%d", a[i][j]);
}
```
   (a) 1 2 3 4 5 0 (b) 1 2 3 4 5 junk
   (c) 1 2 3 4 5 5 (d) Run time error
9. Which of the following is a post-test loop?
   (a) do-while (b) for
   (c) if (d) while

10. What is the output of following C code (on a 32-bit machine)?

```
int main()
{
    int x = 10000;
    double y = 56;
    int *p = &x;
    double *q = &y;
    printf("p and q are %d and %d", sizeof(p), sizeof(q));
    return 0;
}
```

(a) p and q are 4 and 4 (b) p and q are 4 and 8
(c) Compiler error (d) p and q are 2 and 8

11. Which of the following is correct statement?
(a) Convert built in data type to user defined type called unboxing.
(b) Convert user defined data type to built-in type called boxing.
(c) if (a&&b||c) makes short circuit evaluation.
(d) Selection type of control flow uses for loop.

12. What is a Loop in programming language?
(a) A Loop is a block of code that is executed repeatedly as long as a condition is satisfied.
(b) A Loop is a block of code that is executed only once if the condition is satisfied.
(c) A Loop is a block of code that is executed more than 2 times if the condition is satisfied.
(d) None of the above

13. A CONTINUE statement inside a Loop like WHILE, FOR, DO-WHILE and Enhanced-FOR causes the program execution _____ the loop.
(a) Skip
(b) Skip present iteration and continue with next iteration of the loop.
(c) Exit
(d) None

14. The continue statement cannot be used with _____.
(a) for (b) while
(c) do while (d) switch

15. Which keyword can be used for coming out of recursion?
(a) return (b) break
(c) exit (d) Both (a) and (b)

16. Which of the following is an invalid if-else statement?
(a) if (if (a == 1)){} (b) if (a){}
(c) if ((char) a){} (d) if (func1 (a)){}

Answers

1. (b)	2. (a)	3. (a)	4. (a)	5. (a)	6. (a)	7. (b)	8. (a)	9. (a)	10. (a)
11. (c)	12. (a)	13. (b)	14. (d)	15. (a)	16. (a)				

Q.II Answer the following questions in short.

1. Name eight major categories of control-flow mechanisms.
2. What is the Cambridge Polish notation?
3. Name two programming languages that use Postfix notation.
4. What does it mean for an expression to be referentially transparent?
5. What is an L-value and R-value?
6. Define the Orthogonality.
7. What is a side effect?
8. Why initialization required?
9. What are aggregates?
10. Explain the notion of definite assignment in Java and C#.
11. Does C use enumerated controlled loop?
12. What is a container (a collection)?
13. What is the true iterator?
14. Give an example in which a mid-test loop used.
15. What is memorization & promise?
16. Justify True/False, "Short circuit evaluation can save time".

Q.III Answer the following questions.

1. Explain the difference between Prefix, Infix, and Postfix notation.
2. What is the difference between a value model of variables and a reference model of variables?
3. Differentiate between the applicative & the normal order evaluation.
4. What are the categories of control flow?
5. What is a short-circuit Boolean evaluation? Explain with its advantages.
6. Explain the Enumeration controlled loop.

7. Explain the logically controlled loop.
8. Remove tail recursion:

```
int foo(int n)
{
   if(n==0)
      return 1;
   if(n%2==0)

      return 2*foo(n/2);
   else
      return foo(n-1);
}
```

9. What will be output of following code using Applicative-order and Normal-order?

```
#define max(a,b) a>b?a:b
int min(int a,int b)
{
   return a<b?a:b;
}
main()
{
   int p,q,r,s,u,v;
   p=10,q=20,r=30,s=40
   u=max(p++,q++);
   v=min(r++,s++);
   printf(p,q,r,s,u,v);
}
```

■■■

CHAPTER 5

Subprograms and Implementing Subprograms

Contents

Objectives ...

After learning this chapter you will be cble:

- To explore the design of subprograms.
- To understand different parameter-passing methods, local referencing environments.
- To study the concept of overloading of subprograms, generic subprograms.
- To learn the aliasing and problematic side effects that is associated with subprograms.
- To understand the concept of co-routines.
- To explore the implementation of subprograms.
- To get knowledge of how subprogram linkage works.
- To learn the static chain method of accessing nonlocals in static-scoped languages.
- To study techniques for implementing blocks.
- To learn several methods of implementing nonlocal variable access in a dynamic-scoped language.

5.1 INTRODUCTION

- **Subprogram** is a sequence of instructions; execution is done from one or more remote locations in a program.
- Expecting that after complete execution of subprogram, execution continues from the instruction which comes after the subprogram invocation.
- A computation sequence is named and packaged. Whenever computation sequence is required name is used.
- Subprogram is also called as subroutines, procedures and functions in High-level languages.
- Actually, **Function** is a subroutine that returns a value.
- **Procedure** is type of subroutine which is not returning a value.
- Usually procedure is called as a constructors or methods in object-oriented languages.
- Subprograms can have returned values, parameters and local variables in most modern high-level languages.
- Requires subprogram linkage protocols to use subprograms in the assembly language.
- In programming languages, two fundamental abstraction facilities included:
  1. **Process Abstraction:**
     - Emphasized from early days in high-level language.
     - In all programming languages has a central concept in the form of subprograms.
  2. **Data Abstraction:**
     - Emphasized in the 1980s, when people believed that data abstraction is important equally.

5.2 FUNDAMENTALS OF SUBPROGRAMS

5.2.1 General Subprogram Characteristics

- All subprograms excluding the co-routines have characteristics as follows:
  - A single entry point for each subprogram.
  - During called subprogram (callee) execution, the calling program is suspended that means at any given time, only one subprogram will remain in execution.
  - Control always returns to the caller when called execution terminates of a subprogram (callee). Even though some are anonymous, most subprograms have names.

5.2.2 Basic Definitions

A Subprogram Definition:

- Describes the interface to and the actions of the subprogram abstraction.
- Function definitions are executable in Python, non-executable in all other languages.

A Subprogram Call:

- It is the explicit request that the specific subprogram be executed.

An Active Subprogram:

- A subprogram when called has begun its execution but not yet completed that execution.

A Subprogram Header:

- This is the first part of the definition provides various purposes.
- It provides subprogram name, the kind of subprogram, and the list of formal parameters. For example, `def sub1 parameters`
  where, `sub1` is the header of a Python subprogram.
- In Fortran, it written as,
  `subroutine sub1(parameters)`
- In Ada, `procedure sub1(parameters)`
- In Ruby, subprogram headers begin with **def.**
- In JavaScript, the subprogram header begins with **function.**
- In C, the header of a function named `sub1` will be, `void sub1(parameters)`
- Here, the reserved word `void` means that the subprogram does not return a value.

The Subprograms Body: It defines actions.

- The body of a subprogram is delimited by braces in C-based languages (e.g. JavaScript).
- An end statement terminates the body of a subprogram in Ruby.
- The statements in the body must be indented and the end of the body is indicated by the first statement that is not indented in Python function.

The Parameter Profile:

- It is also known the signature.
- It contains the order, number and parameters types of a subprogram.

The Protocol:

- The Protocol of a subprogram is a subprogram's parameter profile and its return type if it is a function.
  - Types are defined by the subprogram's protocol if subprograms have types.
  - In C and C++, function declarations are often called prototypes. In header files declarations are placed.
  - Subprogram declarations do not include their body but provide the protocol of subprogram.
  - Excluding C and C++, in many other languages no need to declare subprograms because there are no requirement those subprograms is defined before calling.

5.2.3 Parameter and its Types

- Subprograms describe computations.
- A non-method subprogram can access data to process in two ways: through direct access to non-local variables (visible in the sub-program but declared elsewhere) or through parameter passing.
- Data passed through parameters are accessed through names that are local to the subprogram.
- Parameter passing is more flexible than direct access to non-local variables.

A Formal Parameter:

- These Parameters are listed in the subprogram header.
- Variables are not in the usual sense so sometimes called as *dummy variables*.
- When the subprogram is called, mostly bound to storage through some other program variables.

An Actual Parameter:

- A list of parameters to be bound to the formal parameters and name of the subprogram included in subprograms call statements. Such parameters are called Actual parameters.
- The formal and actual parameters have different restrictions on their forms, and their uses are also different.

Actual/Formal Parameter Correspondence:

- Binding between formal parameters and actual parameters are done in the following ways:
  1. **Positional Parameters:**
     - By position the binding or correspondence of actual parameters to formal parameters can be done.

     - The first formal parameter is bound to the first actual parameter and so on.
     - This is a safe and effective method when parameter lists are relatively short.
  2. **Keyword Parameters:**
     - Specified with the actual parameter in a call, the name of the formal parameter to which an actual parameter is to be bound.
     - Here, value default with parameter is set.

     For example, `f(a,b,c=3)`. Here we can call as, `f(a,b)`.

Advantage:

- In the actual parameter list, parameters can appear in any order.

Disadvantage:

- The names of formal parameters must be known to the user of the subprogram.

Formal Parameter Default Values:

- In certain languages like Python, Ruby, Fortran 95+, PHP, C++ and Ada, formal parameters can have default values.
- If actual parameter is not passed then a default value of formal parameter is used.
- As parameters are associated by position, default parameters must appear last in C++. All remaining formal parameters must have default values, once a default parameter is omitted in a call.
- C# methods accept a variable number of parameters, as long as they are of the same type. Formal parameters are specified with the **params** modifier by the method.

5.2.4 Procedures and Functions

- There are two main categories of subprograms: Procedures and Functions.

Procedures:

- Procedures are collection of statements that provide parameterized Computations. These computations are passed by single call statements.
- In Procedures, results can be produced in the calling program unit using two methods:
  - The procedure can change the variables which are still visible in both the procedure and the calling program unit but formal parameters can not.
  - Formal parameters of the subprogram allow the transfer of data to the caller can be changed.

Functions:

- Functions structurally look like procedures.
- It provides user-defined operators, semantically modeled on mathematical functions.
- If a function is a faithful model, it produces no side effects. That means it modifies neither any variables defined outside the function nor its parameters.
- In practice, functions have side effects in most programming languages.
- The methods of Java, C++, and C# are syntactically similar to the functions of C.
- Functions are called by their names in expressions along with the required actual parameters.

5.3 DESIGN ISSUES FOR SUBPROGRAMS

- In programming languages, subprograms are complex structures. A lengthy list of issues is involved in their design. Some of them are listed below:
  - Which parameter passing method/ methods are provided?
  - Whether parameter types checked or not?
  - Whether functional side–effects are allowed or not?
  - Whether local variables are allocated statically or dynamically?
  - What types of values can be returned from the functions?
  - How many values function can be return?
  - Are subprograms definitions appear in other subprogram definitions?
  - Whether subprograms are overloaded?
  - Is the subprogram allowed to be generic?
  - Are closures supported, when the language allows nested subprograms?
  - What is the referencing environment of a passed subprogram, if subprograms can be passed as parameters and subprograms can be nested?
  - Whether the types of the actual parameters checked against the types of the Formal parameters?

5.4 LOCAL REFERENCING ENVIRONMENTS

Definition of the Referencing Environment of a statement:

- The collection of all names visible in the statements is called as the referencing environment of a statement.

1. Local Variables:

- Local variables are defined inside subprograms.
- Usually the body of the subprogram in which they are defined is the Scope.
- Stack dynamic or Static local variables are when program begins execution bound to storage and when execution terminates unbound.
- In a Static-scoped language, the referencing environment is:
  - Local variables and also visible variables in all scopes that are enclosed.
- If execution has begun but has not yet terminated then subprogram is active.
- In a dynamic-scoped language, the referencing environment is:
  - Local variables + Visible variables in all subprograms which are active.

2. Stack-dynamic Variables:

- Local variables can be stack-dynamic (means bound to storage).

Advantages of using Stack-dynamic:

- It supports recursion.
- Among some subprograms, storage for locals is shared.
- Flexibility.

Disadvantages:

- Time required for Allocation/de-allocation, initialization is more.

- Indirect addressing can be "determined during the execution."
- Data values of local variables between calls can not be retained, subprograms cannot be history sensitive.

3. Static variables:

- Local variables can be static.

Advantages of using static variables:

- As no indirection, static local variables can be accessed faster i.e. more efficient.
- For allocation and de-allocation no run-time overhead.
- Subprograms might be history-sensitive.

Disadvantages:

- Inability to support recursion.
- Storage can not be shared with the local variables of other inactive subprograms.
- Local variables are stack-dynamic in C and C++ functions, unless specifically declared to be static.

Example:

```
int (int A[ ], int alength)
{
    static int total = 0; // total is static variable
    int i; // i is stack-dynamic
    for (i = 0; i< alength; i++)
    total += A[i];
    return total;
}
```

In the above example, the variable **'total'** is static and variable **i** is stack-dynamic.

- The methods of C#, C++ and Java and Ada subprograms have only stack-dynamic local variables.
- All local variables are stack-dynamic in Python methods.
- Closure is bundled a reference to a subroutine and its referencing environment to the subroutine.

5.5 PARAMETER PASSING METHODS

- Parameter passing methods are ways in which parameters are transmitted to and/or from called subprograms.

5.5.1 Semantic Models of Parameter Passing

- Formal parameters are characterized in distinct semantics models as follows:
  - **In mode:** Receives data from the corresponding actual parameter.
  - **Out mode:** Transmits data to the actual parameter.
  - **In-Out mode:** This mode can do both operations that means they receive data from and transmit data to the corresponding actual parameters.

- In Parameter transmission, there are two conceptual models of how data transfers take place:
  - Move a value physically means an actual value is copied (to the caller, to the callee, or both ways).
    OR
  - To a value, an access path is moved means transmitted an access path to a value.
- Mostly, the access path is a reference or a simple pointer.
- Figure below illustrates the three semantics of parameter passing when values are copied.

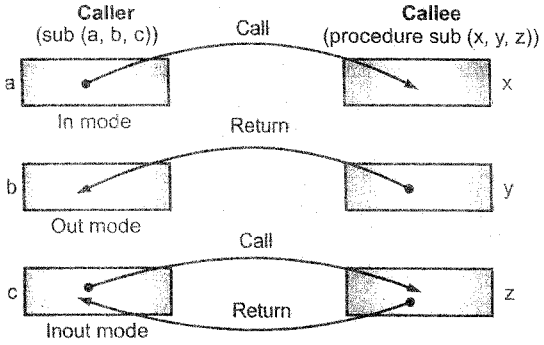


Fig. 5.1: Models of Parameter Passing

5.5.2 Parameter Passing Models

- Language designers developed a variety of models to guide the implementation of the basic parameter transmission modes.
- Consider the Code segment below:

```
main()
{
   int a=10,b=20;
   Foo(a++,b++);
   print(a, b);
}
Foo(int x, int y)
{
   x++;
   y--;
   x+=10;
   y-=10;
   x--;
   y++;
}
```


- Five categories of the parameter passing model are:
  1. Pass-by-Value (In Mode)
  2. Pass-by-Result (Out Mode)
  3. Pass-by-Value-Result (In-Out Mode)
  4. Pass-by-Reference (In-Out Mode)
  5. Pass-by-Name (In-Out Mode)

5.5.2.1 Pass-by-Value (In Mode)

- The value of the actual parameter is used to initialize the corresponding formal parameter when a parameter is passed by value.
- Implementing in-mode semantics as parameters acts as a local variable in the subprograms.
- Implemented by copying normally.
- It can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy).

Advantage:
- For scalars, in both linkage cost and access time it is fast.

Disadvantages (if by physical move):
- For the formal parameter, additional storage is required (stored twice) .
- The actual move can be costly (for large parameters) as actual parameter must be copied to the storage area for the corresponding formal parameter.

Disadvantages (if by access path method):
- In the called subprogram, the value must be write-protected.
- Access cost is more (indirect addressing)

- When copies are used, additional storage is required.
- Storage and copy operations can be costly.
- Consider code given above, using call by value output of code is,

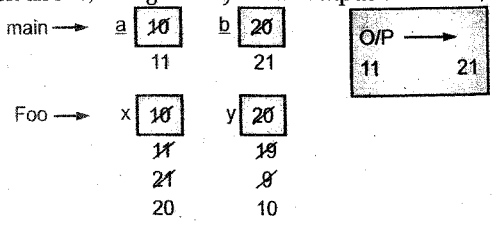


Fig. 5.2: Output of code in Section 5.5.2 using Pass-by-Value Model

5.5.2.2 Pass-by-Result (Out Mode)

- Pass-by-result is an implementation model for out-mode parameters.
- When a parameter is passed by result, value is not given to the subprogram.
- When control is returned to the caller by a physical move, the corresponding formal parameter acts as a local variable and to caller's actual parameter its value is transmitted.

- Advantages and disadvantages are the same as pass-by-value, plus some additional disadvantages.
- It requires extra storage location and copy operation.

Potential Problem:

```
Add(V1,V1);
```

- In Add(), assume two formal parameters have different names and assigned different values.
- Whichever formal parameter is copied to their corresponding actual parameter last becomes the value of V1. This means whichever formal parameter is copied back will represent the current value of V1.

5.5.2.3 Pass-by-Value-Result (In-Out Mode)

- Pass-by-value-result is an implementation model for In-Out mode parameters in which actual values are copied.
- It is a combination of Pass-by-Result and Pass-by-Value.
- At subprogram, entry the actual parameter is copied to the formal parameter and at subprogram termination copied back, so sometimes called as **Pass-by-Copy**.
- To initialize the formal parameter, value of the actual parameter is used, which acts as a local variable.
- Local storage for formal parameters.
- The **advantages** are relative to Pass-by-Reference.

Disadvantages:

- Shares with pass-by-result and pass-by-value, requires multiple storage for parameters and time for copying values.
- Shares with pass-by-result, the problems associated with the order of actual parameters assignment.

- Output of the above code using this model can be shown as:

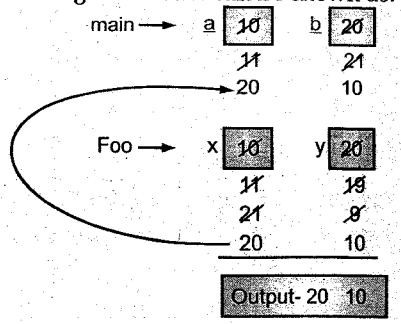


Fig. 5.3: Output of code in Section 5.5.2 using Pass-by-Value-Result Model

5.5.2.4 Pass-by-Reference (In-Out Mode)

- Pass-by-reference is the second implementation model for In-Out mode parameters.
- Pass an access path or sometimes an address to the called subprogram rather than copying data values.
- It provides access path to the cell which stores the actual parameter.
- With the called subprogram, actual parameter is shared. It is also called *pass-by-sharing*.

Advantage:

- In terms of time and space, the passing process is efficient as no copying and no duplicated storage is required.

Disadvantages:

- Slower access to formal parameters compared to pass-by-value as an additional level of indirect addressing is required.
- Possibilities of unwanted side effects (collisions).
- Unwanted aliases (access broadened) can be created as in C++,

```
void Sum( int &I, int &J)
```

- If the call to Sum passes the same variable twice, as follows `Sum(K,K)` then I and J in Sum will be aliases.
- Changes made to the actual parameter may lead to inadvertent and erroneous changes.

- Using call by reference output of above code is,

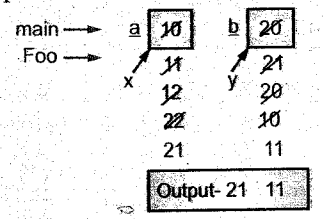


Fig. 5.4: Output of code in Section 5.5.2 using Pass-by-Reference Model

5.5.2.5 Pass-by-Name (In-Out Mode)

- Pass-by-Name does not correspond to a single implementation model for In-Out mode parameter transmission.
- When parameters are passed by name, the actual parameter is substituted for the corresponding formal parameter in all its occurrences in the subprogram.
- At the time of the call, formal parameters are bound to an access method, and at the time of a reference or assignment, actual binding to a value or address takes place.
- In late binding, this mode allows flexibility.

Disadvantages:

- Parameters are complex to implement and inefficient.
- As adds complexity to the program, lowers readability and reliability.

- Using Pass-by-Name, output of the above code is,

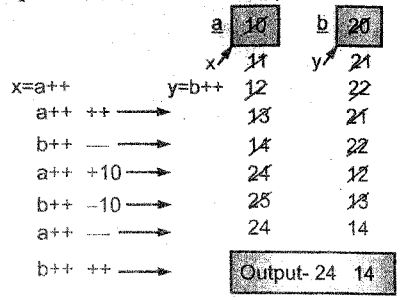


Fig. 5.5: Output of code in Section 5.5.2 using Pass-by-Name Model

5.5.3 Difference between Call by Value and Call by Reference in C

Table 5.1: Call by value vs. Call by reference

Sr. No.	Call by value	Call by reference
1.	A value is passed to the function.	An address of value instead of a value as it is passed to the function.
2.	Changes made in the inner side of the function are limited to the function itself.	Changes made in the inner side of the function make changes also in outer side of the function.
3.	Actual and formal parameters are using different memory locations. So, additional storage is required for the formal parameter.	Actual and formal parameters are using the same memory location.
4.	The values of the actual parameters do not change by changing the formal parameters.	The values of the actual parameters do change by changing the formal parameters.

5.6 PARAMETERS THAT ARE SUBPROGRAMS

- Convenient to pass subprogram names as parameters to other subprograms sometimes.
- Issues are:
  - Whether parameter types checked or not?
  - What is the correct referencing environment sent as a parameter for a subprogram?

Parameters that are Subprogram Names: Parameter Type Checking

- In C and C++, pointers to functions can be passed as parameters but functions cannot be passed. Parameters can be type-checked as their types include the types of the parameters.

- Parameter type Checks in FORTRAN 95.
- Similar alternative is provided through Ada's generic facility as later versions of Pascal and Ada do not allow subprogram parameters.
- In Java, method names can not be passed as parameters.

Parameters that are Subprogram Names: Referencing Environment

- In languages which allow nested subprograms, occur issues related to subprogram names that are passed as parameters.
- The question is: For executing the passed subprogram what referencing environment should be used?

- There are following three choices of bindings:

(**Note:** We have already learned each category in chapter 2 in brief.)

1. **Shallow binding**:
   - The call statements environment which enacts the passed Subprograms.
   - For dynamic-scoped languages it is most natural.
2. **Deep binding**:
   - The definition of the passed subprograms environment.
   - For statically scoped languages, it is most natural.
3. **Ad-hoc binding**:
   - The call statements environment which passed the subprogram.
   - Ad-hoc binding has never been used because, one can assume that the environment in which the procedure appears as a parameter has no natural connection to the passed subprogram.

Example: In Java,

```
function add1( )
{
   var A;
   function add2( )
   {
      alert(A);
      // Creates a dialog box with the value of A
   };
   function add3( )
   {
      var A;
      A = 6;
      add4(add2);
   };
```

```
    function add4(addA )
    {
       var A;
       A = 8;
       addA( );
    };
    A = 2;
    add3( );
};
```

- Let us see the execution of `add2()` which is called in `add4()`.
  - **Shallow Binding:** The referencing environment of execution is that of `add4()`, so the reference to `A` in `add2()`, bound to the local `A` in `add4()` therefore output is 8.
  - **Deep Binding:** The referencing environment of add2's execution is that of add1. Therefore, the reference to `A` in `add2()` is bound to the local `A` in `add1()`, therefore the output is 2.
  - **Ad-hoc Binding**: Binding is to the local `A` in `add3()`. Therefore, the output is 6.
- For static-scoped languages, Shallow binding is inappropriate with nested subprograms.

5.7 OVERLOADED SUBPROGRAMS

- **An overloaded subprogram**: In the same referencing environment, a subprogram having the same name as another subprogram.
- It has a unique protocol for overloaded subprograms in every version.
- It must be different from other subprograms by the order, number or types of parameters or its return if a function.
- The actual parameter list determines the meaning of a call to an overloaded subprogram.
- Predefined overloaded subprograms are included in C++, Java, C#, and Ada.
- To disambiguate calls, the return type of an overloaded function can be used in Ada (two overloaded functions might have the same parameters).
- Writing multiple versions of subprograms with the same name is allowed to users in Ada, Java, C++, and C#.
- Overloaded subprograms having default parameters might contain ambiguous subprogram calls.

Examples:

1.
```
void sum(float S = 0.0);
void sum( );
   ...
sum( ); // A compilation error as the call is ambiguous
```


2. Some languages allow function overloading. It can be in two variations as:

(i) According to Parameters:

```
void F(int a, int b)
{
   //code for integer operations
}
void F(float a, float b)
{
   //code for float operations
}
F(10,20)
F(10.2,40.3)
```

(ii) According to Return type:

```
void F(int a,int b)
{
   print a;
}
int F(int a,int b)
{
   return a;
}
main()
{
   F(10,20);
   int X = F(10,20);
}
```

5.8 GENERIC SUBPROGRAMS

- There is no need to write four different sort subprograms to sort four arrays which differ in element type only by the programmer.
- Parameters of different types on different activations taken by a polymorphic or a generic subprogram.

(Note: We have already discussed these points in Chapter 2.)

1. **Ad-hoc Polymorphism:** A particular kind of polymorphism is provided by overloaded subprograms.
2. **Parametric Polymorphism:** It is provided by a subprogram that takes a generic parameter used in a type expression that describes the type of the parameters of the subprogram provides parametric polymorphism.

3. **Subtype Polymorphism:** Any object of type T or any type derived from T can be accessed by variable of type T (OOP languages).

5.8.1 Generic Functions in C++

- Generic Functions have the descriptive name of template functions.
- Implicitly created versions of a generic subprogram, when the subprogram is named in a call or when address is used with and operator.
- Generic Functions preceded by a template clause that lists the generic variables, which can be class names or type names.
- **Example:**

```
template<class Type>
Type maximum(Type X, Type Y)
{
   return X > Y ? X: Y;
}
```

- This function template can be instantiated for any type for which operator > is defined.

```
int maximum (int X, int Y)
{
   return X > Y ? X: Y;
}
```

- **Example:** Generic sort subprogram.

```
template<class Type>
void gen_sort (Type Slist[ ],int l)
{
   int first, last;
   Type temp;
      for (first = 0, first < l-2; first++)
      for (last = first + 1; last < l- 1; last++)
      if (Slist [first] > Slist [last])
         {
            temp = Slist [first];
            Slist[first] = Slist[last];
            Slist[last] = temp;
         } // end for last
} // end for generic
```


5.8.2 Generic Functions in Java

- In Java 5.0, Generic parameters must be classes and generic methods are instantiated just once as truly generic methods.
- Restrictions can be specified on the range of classes which can be passed to the generic method as generic Parameters.
- Wildcard types are generic parameters.
- For example,

```
public static <A> A doIt(A[] list)
{ ... }
```

- The parameter: An array of generic elements (A is the name of the type).
- A call:

```
doIt<String>(myList);
```

- Generic parameters have bounds:

```
public static <A extends Comparable> A
doIt(A[] list)
{ ... }
```

- The generic type must be of a class, implements the Comparable interface.
- **Wildcard Types:** Collection<?> is a wildcard type for collection classes.

```
void dispCollection(Collection<?>C)
{
   for (Object O: C)
   {
      System.out.println(O);
   }
}
```

- It works for any collection class.

5.9 DESIGN ISSUES FOR FUNCTIONS

Design issues for functions are:

- Are side effects allowed in functions?
- Parameters should always be in-mode to reduce side effect (like Ada).
- How many values can be returned?
- In functions which types of returned values are allowed?
- Most imperative languages restrict the return types.

5.9.1 Functional Side Effects

- Parameters to functions should always be in-mode parameters as there is a problem of side effects of functions called in expressions.

- These parameters have only in-mode formal parameters in Ada functions.
- It prevents functional side effects through its parameters or aliasing of parameters and global.
- Functions can have either Pass-by-Value or Pass-by-Reference parameters in most languages, which allow functions that cause side effects and aliasing.

5.9.2 Types of Returned Values

- In C, any type to be returned except arrays and functions by function.
- Ada subprograms allow any return type. In Ada, as subprograms are not types therefore can not be returned.
- C++ is like C but its function also returns user-defined types or classes.
- In JavaScript, functions can be passed as parameters and returned from functions.
- No functions in Java and C#. In Java and C# methods can have any type but can not be returned.
- In Python and Ruby, methods are treated as first-class objects, so can be returned, as well as any other class.
- In Lua, functions return multiple values.

5.10 USER-DEFINED OVERLOADED OPERATORS

- Nearby all programming languages have overloaded operators.
- Operators can be overloaded in Ada, C++, Python and Ruby by users (Not in Java).
- How much operator overloading is good, or can have too much?
- Example in Ada,

```
function "*"(X,Y: in Vec_Type): return Integer is
   total: Integer:= 0;
   begin
   for I in X ' range loop
   total:= total + X(I) * Y(I)
   end loop
   return total;
   end "*";
      ...
   z= x * y; -- x, y, and z are of type Vec_Type
```

- A Python Example:

```
def_sum_ (First, Second):
returnComplex(First.real + Second.real, First.imag + Second.imag)
```

Use:

- To compute a +b, a._sum_(b).

5.11 CO-ROUTINES

5.11.1 Basic Concept

- A co-routine is a subprogram that has multiple entries and controls them itself. It is also called as symmetric control because called and caller subroutines are on a more equal base.
- It has to maintain their status between activation.
- A call is named a **Resume** in co-routine.
- Must be history sensitive and having static local variables.
- Secondary executions begin at points other than its beginning for co-routine.
- In the co-routine, first resume of a co-routine is at its beginning, but subsequent calls enter at the point just after the last executed statement.
- Resume each other repeatedly, mostly forever.
- Program unit's quasi-concurrent execution is provided. Execution is not overlapped but interleaved.
- Co-routines can be executed by switching from one to another.
- Execution control may transfer from procedure A to B, A is master routine. This shown in the following figure,

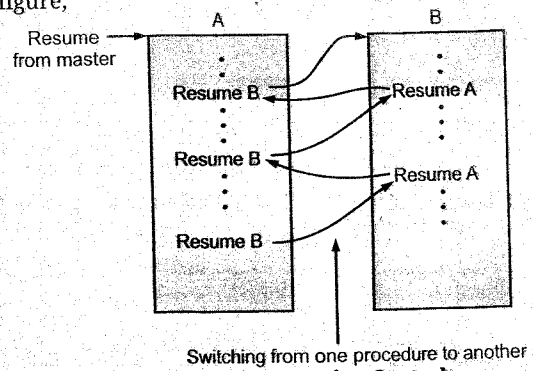


Fig. 5.6: Possible Execution Controls

- Execution control may transfer from procedure B to A, B is master routine.

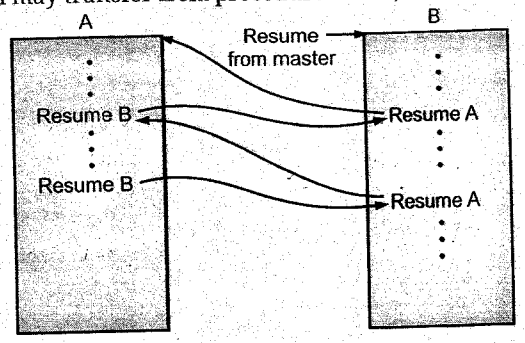


Fig. 5.7: Possible Execution Controls

- Sometimes control transfer can have behavior as loop as it continuously transfers its control within co-routine, which is shown in following Fig. 5.8.

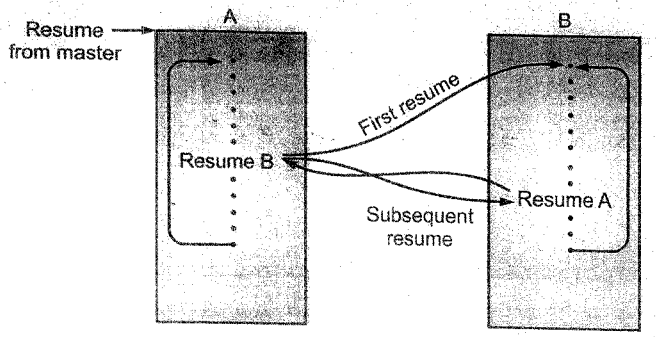


Fig. 5.8: Possible Execution Controls with Loops

- **Transfer:** To switch from one co-routine to another the system at run-time must change the PC, stack, and register information. This is handled by Transfer.

5.11.2 Allocation

- As co-routines are concurrent they can't share a single stack as we know subroutine calls and from that returns which are not LIFO.
- For this purpose, at run-time system uses data structure for stack allocation known as a *cactus stack* to allow sharing.
- Consider nesting subroutines:

Fig. 5.9: Nesting Subroutines

- Following figure shows how *cactus stack* look like for different co-routines with different calling sequences,

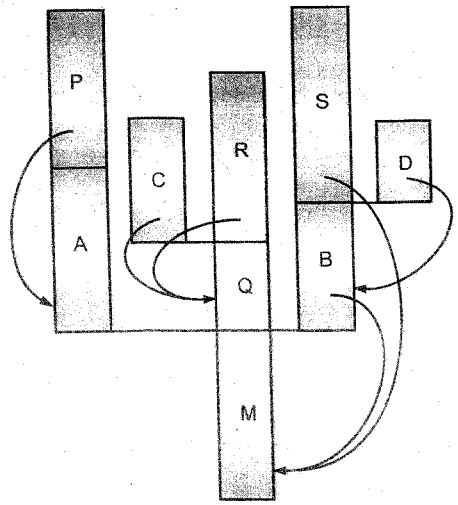


Fig. 5.10: Different co-routines with different Calling Sequences

- In this Figure, arrows denote static links.

5.12 IMPLEMENTING SUBPROGRAMS

5.12.1 The General Semantics of Calls and Returns

- **Subprogram Linkage**: The subprogram call and return operations of a language are together called **Subprogram Linkage**.
- The subroutines implementation must be based on the semantics of the subroutine linkage.
- In a typical language, subprogram call has numerous actions associated with it.

General semantics of Subprogram Calls:

- Save the execution status of the current program unit (or calling program).
- Pass the parameters using Parameter passing methods.
- Local variables have Stack-dynamic allocation.
- Arrange for the return address to the callee.
- To the callee transfers the control.
- Access to nonlocal variables must be arranged if supporting subprogram nesting.

General semantics of Subprogram Returns:

- The current values of parameters are moved to the corresponding actual parameters, if there is Pass-by-Value-Result or out-mode parameters.
- In mode and In-Out mode parameters values must be returned.
- The return value is moved to a place accessible to the caller.

- Restored the caller's execution status.
- The control is transferred to the caller back.
- Stack-dynamic locals deallocation.

5.12.2 Implementing "Simple" Subprograms

- **Simple Subprograms**: Subprograms cannot be nested and Static local variables.
- For a *simple* subprogram, the semantics of a call requires the following actions:
  - The execution status of the caller should be saved.
  - The parameter-passing process should be carried out.
  - To the callee passing the return address.
  - Control should be transferred to the callee.
- For a *simple* subprogram, the semantics of a return requires the following actions:
  - If Pass-by-Value-Result or Out mode parameters are used, move the current values of those parameters to their corresponding actual parameters.
  - Move the function return value to a place accessible to the caller if a function is used.
  - The execution status is restored of the caller.
  - The Control is transferred back to the caller.
  - The call and return actions require storage for the following:
    - Status information of the caller
    - Parameters
    - Return address
    - Functions return value (if it is a function)
    - Temporaries
- These, along with the local variables and the subprogram code, form the set of information that a subprogram needs to execute and returns control to the caller.
- A *simple* subprogram consists of two separate parts:
  1. **Code:** The actual code of the subprogram is constant (instruction space).
  2. **Non code:** When the subprogram is executed, the local variables and data can change (Data space).
- Both Code and Non code parts have fixed sizes.

Activation Record (AR):

(Note: We have already learned AR in Chapter 2.)

- Only during the activation or execution of the subprogram, data it describes are relevant.
- An Activation records form is static.
- A concrete example of an activation record is an **Activation Record Instance** corresponding to one execution. (For a particular subprogram activation collection of data).

- There can be only one active version of a given subprogram at a time as languages with *simple* subprograms do not support recursion.
- For a subprogram, there can be only a single instance of the activation record.
- A possible layout for activation records is shown in the following Fig. 5.11.

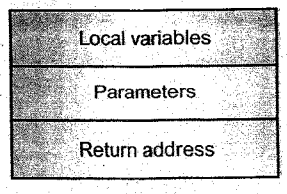


Fig. 5.11: Layout for Activation Records

- Activation records can be statically allocated as an activation record instance has a fixed size for a *simple* subprogram.
- Activation records could be attached to the subprograms code part.
- The following figure shows a program consisting of the main program and three subprograms: A, B, and C.

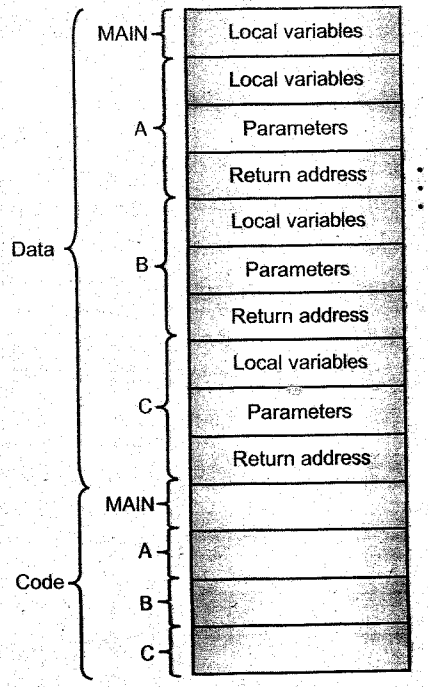


Fig. 5.12: A Program consisting of a main program and three subprograms

- By the compiler complete programs construction is not done entirely in the above case.

- Independent compilation of MAIN, A, B, and C may take place at different time individually (on different days or even in different years).
- When each unit is compiled the machine code along with a list of references to external subprograms is written to a file.
- By the **linker**, the executable program is put together which is part of the Operating system.
- The linker was called for MAIN finds and loads all referenced subroutines, including code and activation records, into memory for programs A, B and C with the code for MAIN.
- Sets the target addresses to subroutines entry addresses of calls.

5.12.3 Implementing Subprograms with Stack-Dynamic Local Variables

- Support for recursion is the most important advantage of Stack-dynamic local variables.

More Complex Activation Records:

- Subprogram linkage in languages using stack-dynamic local variables is more complex than the linkage of *simple* subprograms for the following reasons:
  - For the implicit allocation and deallocation of local variables compiler must generate code.
  - Must support Recursion (adds the possibility of multiple simultaneous activations of a subprogram) which means can be more than one instance of a subprogram at a given time with one or more recursive calls and one call from outside the subprogram.
  - Recursion requires multiple instances of activation records, one for each subprogram activation that exists at the same time.
  - Requires own copy of the formal parameters and the dynamically allocated local variables, along with the return address for each activation.
- At compile time, the format of an activation record for a given subprogram is known in most languages.
- For activation records the size is known as all local data is of fixed size in many cases.
- The activation record format is static and its size may be dynamic.
- In languages having stack-dynamic local variables, when subprogram is called activation record instances must be created dynamically.
- The activation record for such a language is shown in the following figure.

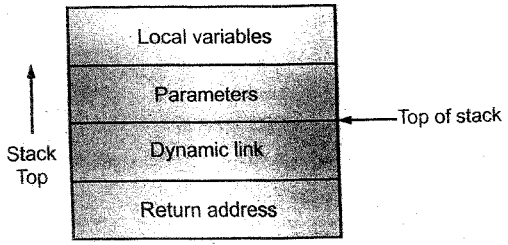


Fig. 5.13: Activation record for languages having stack-dynamic local variables

- These entries must appear first as the return address, dynamic link, and parameters are placed in the activation record instance by the caller.

Return Address:

- Return Address consists of a pointer to the code segment of the caller and an offset address in that code segment of the instruction following the call.

Dynamic Link:

- Dynamic Link points to the top of an instance of the activation record of the caller.
- When the procedure completes its execution, dynamic link is used in the destruction of the current activation record instance in static-scoped languages.
- To the value of the old dynamic link stack top is set.
- The values or addresses provided by the caller are the actual parameters in the activation record.
- It resides on the run-time stack, activation record instances.
- By the run-time system, the Environment Pointer (EP) must be maintained.
- It always points at the base of the activation record instance of the currently executing program unit.

Local Scalar Variables:

- These variables are bound to storage within an activation record instance.

Local Structure Variables:

- Only descriptors and a pointer to storage are part of the activation record. Sometimes these variables are allocated elsewhere.

Local Variables:

- Local variables appear last as allocated and possibly initialized in the called subprogram.
- Consider the following C skeletal function:

```
void add(float total, int part)
{
    int list[4];
    float sum;
    ...
}
```

- The activation record for add() is:

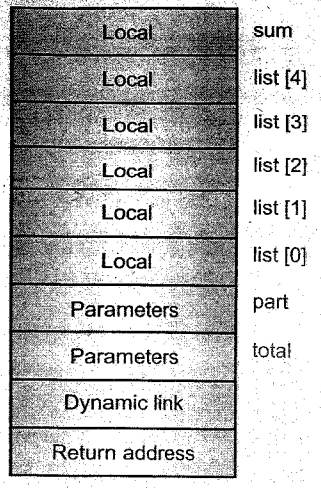


Fig. 5.14: Activation record for add()

- For a subprogram requires the dynamic creation of an instance of the activation record for activating subprogram.
- It is reasonable to create instances of activations records on a stack as the call and return semantics specify the subprogram last called is the first to complete.

Run – time stack:

- Run – time stack is a part of the run-time system.
- On the stack, a new instance of an activation record is created by every subprogram activation whether recursive or non-recursive.
- It provides the required separate copies of the parameters, local variables, and return address.

Example without Recursion:

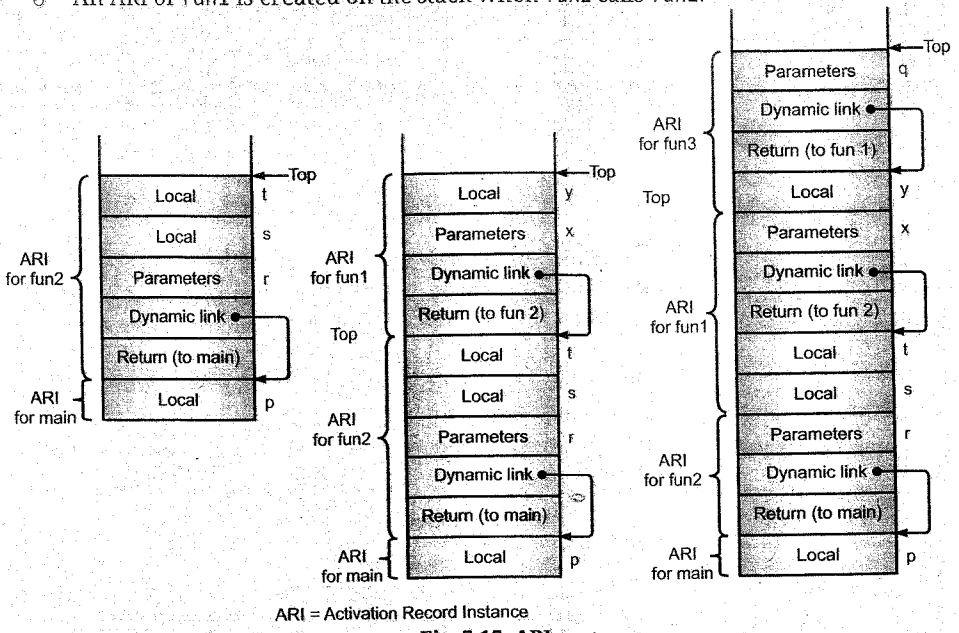
- Consider the following skeletal C program:

```
void fun1(int x) {
                  int y;
                  ...
                  fun3(y);
                  ...
                  }
void fun2(float r)
                  {
                  int s, t;
                  ...
                  fun1(s);
                  ...
                  }
```



```
void fun3(int q) {
                  ...
                  }
void main()       {
                  float p;
                  ...
                  fun2(p);
                  ...
                  }
```

- The sequence of procedure calls in this program is:
  `main calls fun2, fun2 calls fun1, fun1 calls fun3`
- The stack contents are shown in the following figure:
  - As main() calls `fun2()`, only ARI for main and `fun2` are on the stack in first case.
  - An ARI of `fun1` is created on the stack when `fun2` calls `fun1`.



ARI = Activation Record Instance

Fig. 5.15: ARI

  - An ARI of `fun3` is created on the stack when `fun1` calls `fun3`.
  - An ARI of `fun3` is removed from the stack when `fun3's` execution ends, and the dynamic link is used to reset the stack top pointer.
  - When `fun1` and `fun2` terminates, same process takes place.
  - After the return from the call to `fun2` from main, the stack has only the ARI of main.
- In this example, assumed that the stack grows from lower addresses to higher addresses.

Dynamic chain or Call chain:

- This is a collection of dynamic links present in the stack at a given time.
- It represents the dynamic history regarding how execution got to its current position, which is always in the subprogram code and its activation record instance is on the top of the stack.
- **Local_offset:** From the beginning of the activation record of the local scope references to local variables can be represented in the code as offsets.
- Using the order, types, and sizes of variables declared in the subprogram associated with the activation record, the local_offset of a local variable can be determined by the compiler at compile time.
- Assuming all variables take one position in the activation record.
- The local variable declared would be allocated in the activation record **two** positions plus the number of parameters (the first **two** places are for the return address and the dynamic link).
- The second local variable declared would be **one** position nearer the stack top and so forth; e.g. the local_offset of **y** is **3** in **fun1**.
- The local_offset of **s** is **3**; for **t** is **4** in **fun2**.

Recursion:

- Consider the following C program which uses recursion:

```
int fact(int n)
{
   if (n <= 1)
      return 1;
   else
      return (n * fact(n - 1));
}
void main()
{
   int F;
   F = fact(3);
 }
```

- The activation record format for the program is shown below:

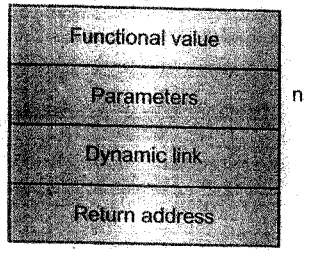
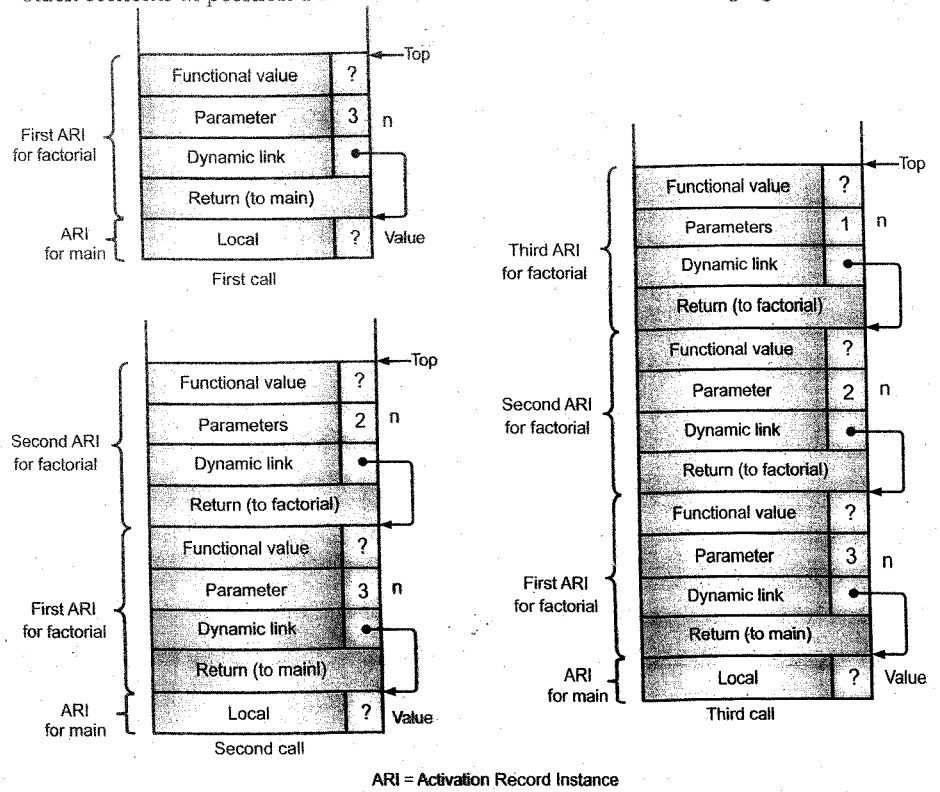


Fig. 5.16: Activation Record Format

- For the returned value of the function, the additional entry in the ARI.
- The content of the stack for the three times that execution reaches position 1 in the function factorial is shown in the following figure.
- One more activation of the function with its functional value undefined is shown.
- The return address to the calling function main is in the first ARI.
- For the recursive calls, others have a return address to the function itself.
- In the following figure, the stack contents for the three times that execution reaches position 2 in the function factorial.
- Position 2 is meant to be the time after the **return** is executed but before the ARI has been removed from the stack.
- Returns 1, the first return from factorial. Therefore, ARI for activation has a value n=1.
- The result of multiplication, 1 is returned to the second activation of factorial to be multiplied by n=2.
- Returns the value 2 to the first activation of factorial to be multiplied by n= 3, produces the result 6 and returned to the first call to factorial in main.
- Stack contents at position 1 in factorial is illustrated in the following figure.



ARI = Activation Record Instance

Fig. 5.17: Stack contents for the Factorial Example

- The stack contents during execution of main and factorial is shown in the following figure.

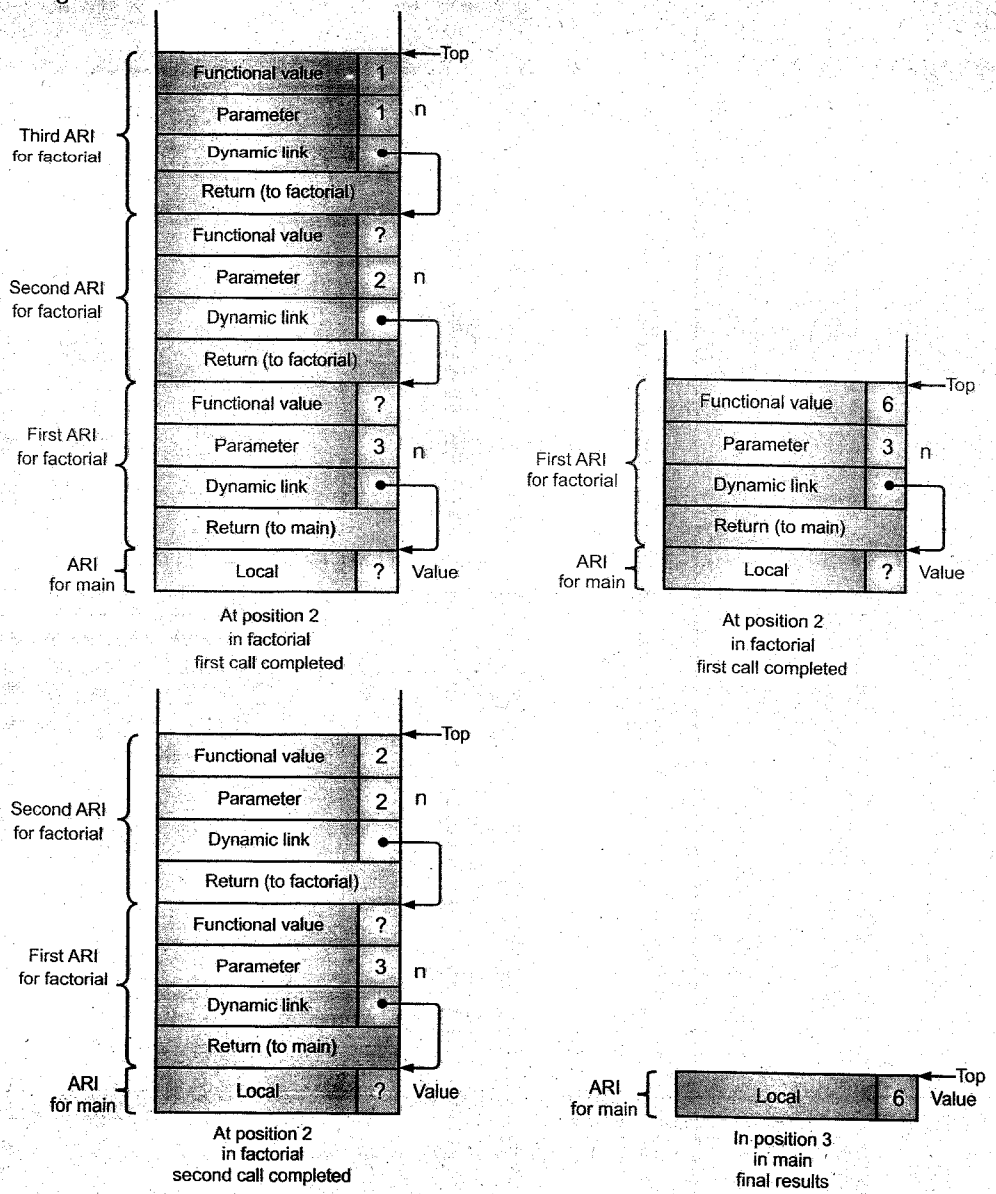


Fig. 5.18: ARI for Factorial Example

5.12.4 Nested Subprograms

(Note: We have learned Nested subprograms in Chapter 2.)

- Some of the static-scoped languages like Fortran 95, Ada, JavaScript use stack-dynamic local variables and allow subprograms to be nested.

1. **Basics:**
   - All variables reside in some activation record instance in the stack which can be non-locally accessed.
   - A non-local reference locating process:
     (i) Find the correct ARI (Activation Record Instance) on the stack where the variable was allocated. It is more challenging and difficult.
     (ii) Determine (Use) the correct offset of the variable within that activation record instance to access it.

Locating a Non-local Reference:

- Correct offset finding is easy.

Process for Locating a Non-local Reference is :

- Searching the correct activation record instance (ARI).
- Variables declared in static ancestor scopes can be accessed and are visible.
- Static semantic rules guarantee that when the reference is made, all non-local variables which can be referenced have been allocated in some activation record instance on the stack.

- Only when all of its static ancestor subprograms are active, a subprogram is callable.
- The correct declaration is the first one found when looking through the enclosing scopes, mostly nested first is dictated by the semantics of non-local references.

2. **Static Chains:**
   - **Static Chain**: A chain of static links that connects certain activation record instances in the stack for an executing subprogram.
   - To implement nonlocal variable access used.
   - **Static Link**: A static scope pointer in an activation record instance for subprogram A points to one of the activation record instances of A's static parent.
   - In the activation record below, the parameters appears the static link.
   - The static chain from an activation record instance connects it to its entire static ancestor.
   - During the execution of a procedure P, the static link of its activation record instance points to an activation of P's static program unit.
   - If there is one then that instance's static link points to P's static grandparent program unit's ARI (activation record instance).

- In order of static parent first, all the static ancestors of an executing subprogram are linked by the static chain.
- To implement the access to non-local variables in static-scoped languages this chain can be used.
- With static links, finding the correct ARI (Activation Record Instance) is simple.
- When a reference is made to a non-local variable, the ARI containing the variable can be found by searching the static chain until a static ancestor ARI is found to contain the variable.
- As the nesting scope is known at compile time, the compiler can determine both non-local references and the length of the static chain must be followed to reach the correct ARI which contains the non-local object.
- A static_depth is an integer associated with a static scope whose value is the depth of nesting of that scope (scope indicating how deeply it is nested in the outermost scope).

Example of static_depth:

- Here, main having static_depth Zero. A and C contains static_depth One and B having static_depth Two.

```
main ------ static_depth = 0
   A  ------ static_depth = 1
      B      -------  static_depth = 2
   C  ------  static_depth = 1
```

- Need is that the length of the static chain should reach the correct ARI for a non-local reference to a variable A is the difference between the static_depth of the procedure containing the reference to A and the static_depth of the procedure containing the declaration for A.
- This difference is called **chain_offset** or the **nesting_depth** of the reference.
- A reference is represented by an ordered integer pair (chain_offset, local_offset).
- Here,
  - **local_offset:** The offset in the activation record of the variable being referenced.
  - **chain_offset:** The number of links to the correct ARI.

```
Procedure X is
   Procedure Y is
      Procedure Z is
      …
      end;  // Z
    …
   end;  // Y
   …
end;  // X
```


- The static_depth of X, Y and Z are 0, 1 and 2.
- If procedure Z references a variable in X, the chain_offset of that reference would be 2 (static_depth of Z- the static_depth of X i.e. 2-0).
- If procedure Z references a variable in Y, the chain_offset of that reference would be 1 (static_depth of Z- the static_depth of Y i.e. 2-1).
- References to locals can be handled using above mechanism, with a chain_offset of 0.

Example of Ada Program:

```
procedure Main_2 is
X: integer;
Procedure Bigsub is
 A, B, C: integer;
procedure Sub1 is
 A, D: integer;
begin{Sub1}
A:= B + C; <----------------------1
 …
end; { Sub1 }
procedure Sub2(X: integer) is
 B, E: integer;
procedure Sub3 is
 C, E: integer;
begin { Sub3 }
 …
 Sub1;
 …
 E:= B + A: <--------------------2
end; { Sub3 }
begin { Sub2 }
 …
 Sub3;
 …
A:= D + E; <----------------------3
end; { Sub2 }
begin { Bigsub }
 …
Sub2(7);
 …
end; { Bigsub }
begin
 …
Bigsub;
 …
end; { Main_2 }
```

- The sequence of procedure calls is as follows:
  `Main_2 calls Bigsub then Bigsub calls Sub2,`
  `Sub2 calls Sub3 and Sub3 calls Sub1`
- The stack situation when execution first arrives at point 1 in the above Ada program is shown in following figure:

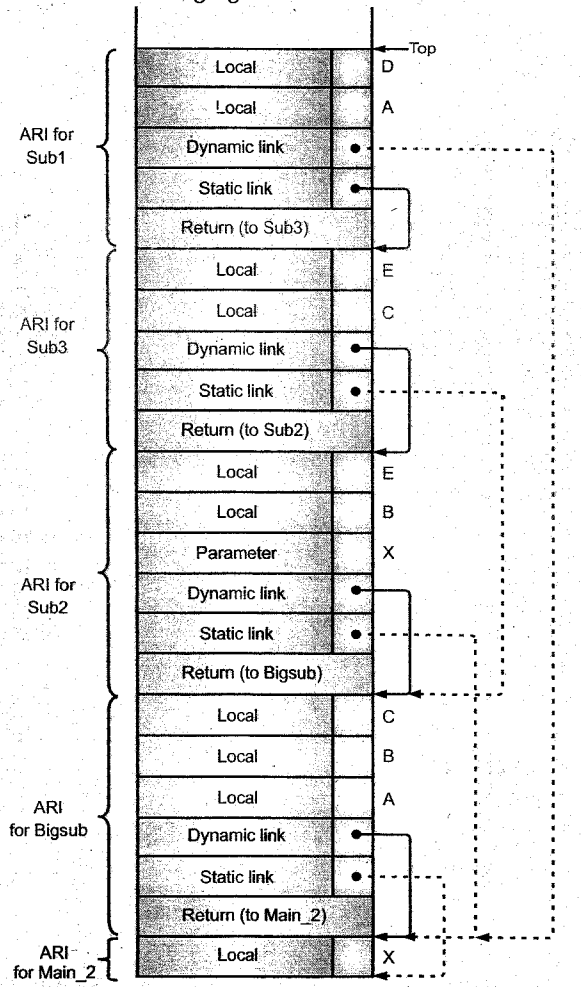


Fig. 5.19: Static Chain

`At position 1 in Sub1:`
`A - (0, 3)`
`B - (1, 4)`
`C - (1, 5)`

`At position 2 in Sub3:`
`E - (0, 4)`
`B - (1, 4)`
`A - (2, 3)`
`At position 3 in Sub2:`
`A - (1, 3)`
`D - an error`
`E - (0, 5)`

5.12.5 Blocks

(Note: We have already learned blocks in Chapter 2.)

- For variables user-specified local scopes we use Blocks.

Example in C:

```
{
    int X;
    X = list [top];
    list [top] = list [bottom];
    list [bottom] =X;
}
```

- In the case of the above example, when control enters the block then lifetime of X begins.
- An advantage of using a local variable like X: cannot interfere with other variable with same name.

Implementing Blocks:

- There are two methods of blocks implementation:
  1. Treat blocks as parameterless subprograms that are always called from the same location. So, there is an activation record for every block. Every time the block is executed, an instance is created.
  2. The maximum amount of storage required for block variables can be statically determined. This amount of space can be allocated after the local variables in the activation record.

5.12.6 Implementing Dynamic Scoping

Deep Access (reference to A):

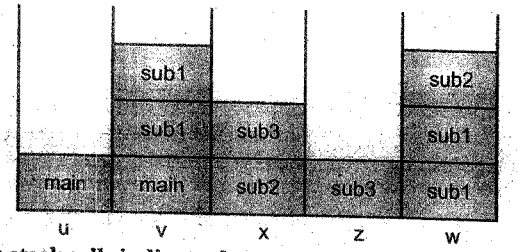
- By searching the activation record instances on the dynamic chain non-local references are found.
- This means following dynamic links until an activation record containing A is found.
- It cannot determine length of the chain statically.
- Activation Record Instances (ARI) must have names of variables.

Shallow Access (reference to A):

- Maintain central table by making entry of each local variable in program.
- Variable's current values live in table not on stack.
- Procedure call:
  - Save current value of A on stack for each local variable A.
- Procedure return:
  - Saved values of variables are restored to table.
- Refers to table all variable access.

Example: Using Shallow Access to Implement Dynamic Scoping:

```
void sub3()
{
   int x, z;
   x = u + v;
   ...
}
void sub2()
{
   int w, x;
   ...
}
void sub1()
{
   int v, w;
   ...
}
void main()
{
   int v, u;
   ...
}
```



(The names in the stack cells indicate the Program units of the Variable declaration)

Fig. 5.20: Shallow Access

SUMMARY

- A subprogram definition describes the actions represented by the subprogram.
- Subprograms can be either functions or procedures.
- Local variables in subprograms can be stack-dynamic or static.
- Some languages allow operator overloading.
- Subprograms can be generic.
- A co-routine is a special subprogram with multiple entries.
- A subprogram definition describes the actions represented by the subprogram.
- Subprograms can be either functions or procedures.
- Local variables in subprograms can be stack-dynamic or static.
- Three basic models of parameter passing: In mode, Out mode, and In-Out mode.
- Subprogram linkage semantics requires many actions by the implementation.
- Simple subprograms have relatively basic actions.
- Stack-dynamic languages are more complex.
- Subprograms with stack-dynamic local variables and nested subprograms have two components: actual code and activation record.
- Activation record instances contain formal parameters and local variables among other things.
- Local variables in static-scoped languages with nested subprograms.
- Access to non-local variables in dynamic-scoped languages can be implemented by use of the dynamic chain or through some central variable table method.

PRACTICE QUESTIONS

Q. I Multiple Choice Questions.

1. Functions and subprograms are both same.
   (a) True (b) False
2. The parameters used at the time of function call are called _____.
   (a) Formal parameters (b) Actual parameters
   (c) Real parameters (d) Complex parameters
3. In which mode parameter lets you pass values to the subprogram being called? It cannot be assigned a value.
   (a) Using the IN mode (b) Using the OUT mode
   (c) Both (a) and (b) (d) None of the above
4. In which parameter mode Formal parameter acts like an initialized variable?
   (a) IN (b) OUT
   (c) IN-OUT (d) None of the above
5. A function is a _____ code.
   (a) Concurrent (b) Sequential
   (c) Concurrent as well as sequential (d) Process oriented

6. How many return arguments can be there in the function?
   (a) 1 (b) 2
   (c) 3 (d) 4
7. If a function has an operator sign as its name, then what will be the purpose of that function?
   (a) Conversion (b) Overloading
   (c) Resolution (d) Define the data type
8. Which of the following statement is correct?
   (a) In pass-by-value, parameters changes actual parameter values.
   (b) In pass-by-value, parameters copied to actual parameter values.
   (c) In pass-by-value, parameters not changes actual parameter values.
   (d) None of the above
9. By default, how the values are passed in C++?
   (a) call by value (b) call by reference
   (c) call by pointer (d) call by object
10. The object _____.
    (a) Can be passed by reference (b) Can be passed by value
    (c) Can be passed by reference or value (d) Can be passed by name
11. Generic data structure can be implemented using _____.
    (a) Function template (b) class template
    (c) File template (d) Inheritance
12. ADA supports the creation of the generic unit by using the keyword_____.
    (a) Template (b) General
    (c) Generic (d) None of these
13. In recursive call, the return address of called function is stored on _____.
    (a) Stack frame (b) Queue
    (c) Tree (d) Cache
14. In the template <class T> declaration of T stands for _____.
    (a) integer data type (b) arbitrary class
    (c) generic data types (d) None of these
15. Which term is not related to co-routine?
    (a) Cactus stack (b) Transfer
    (c) Threads (d) Parameter
16. Bundle with referencing environment and bonding with a reference to the subroutine called _____.
    (a) subroutine (b) co-routine
    (c) block (d) closure

17. Additional storage is required for the formal parameter in ___.
    (a) Call by reference (b) Call by name
    (c) Call by value (d) All the methods

Answers

1. (b)	2. (b)	3. (a)	4. (c)	5. (b)	6. (a)	7. (b)	8. (c)	9. (a)	10. (c)
11. (b)	12. (c)	13. (a)	14. (c)	15. (d)	16. (d)	17. (c)			

Q.II Answer the following questions in short.

1. What are the three general characteristics of subprograms?
2. What is meant by a subprogram is active?
3. What is given in the header of a subprogram?
4. Which languages allow a variable number of parameters?
5. What is a subprogram protocol?
6. What are formal parameters and actual parameters?
7. Write advantages and disadvantages of keyword parameters?
8. What are the design issues for subprograms?
9. What are the advantages and disadvantages of dynamic local variables?
10. What are the advantages and disadvantages of static local variables?
11. What languages allow subprogram definitions to be nested?
12. List out the three semantic models of parameter passing?
13. What are two fundamental design considerations for parameter-passing methods?
14. What are the two issues that arise when subprogram names are parameters?
15. Define shallow and deep binding for referencing environments of subprograms that have been passed as parameters.
16. What is an overloaded subprogram?
17. What is parametric polymorphism?
18. If a Java 5.0 method returns a generic type, what type of object is actually returned?
19. What two languages allow multiple values to be returned from a function?
20. What languages allow the user to overload operators?
21. In what ways are co-routines different from conventional subprograms?
22. What is Transfer?
23. Explain cactus stack.

Q.III Answer the following Questions.

1. Differentiate between a function and a procedure?
2. Explain the semantic models of parameter passing?
3. What are the advantages and the disadvantages of pass-by-value, pass-by-result, pass-by-value-result, and pass-by-reference parameter-passing methods?

4. Describe the ways that aliases can occur with pass-by-reference parameters.
5. In what fundamental ways do the generic parameters to a Java 5.0 generic method differ from those of C++ methods?
6. What are the design issues for functions?
7. Explain pass by value, pass by value-result, pass by name.
8. Differentiate between pass by value and pass by reference.
9. Show output for following code by using Pass by value, Pass by value-result, Pass by name and Pass by value-result.

```
A: int=5;
B: int= 10;
void F(I:int, J:int)
{
   --I=J--;
   print (A+I, B+J);
}
F(A,A+B);
print A;
```

10. Explain blocks in brief.
11. Write note on implementing dynamic scoping.
12. Define the terms:
    (i) Static Chain
    (ii) Static_depth
    (iii) Nesting_depth
    (iv) Chain_offset

■■■